A systems architecture for sensor networks based on hardware/software co-design

Andy Nisbet[1] and Simon Dobson[2]

[1] Manchester Metropolitan University, Manchester UK
a.nisbet@mmu.ac.uk
[2] Department of Computer Science, University College, Dublin IE
simon.dobson@ucd.ie

Abstract. We describe the motivation and design of a novel embedded systems architecture for large networks of small devices, tha canonical example being wireless sensor networks. The architecture differs from previous work in being based explicitly on a hardware/software co-design approach centred around the deployment of novel programming language constructs directly onto hardware in order to improve optimisation and expressibility. The programming interface enables the dynamic download and execution of domain-specific code to facilitate the development of context aware pervasive computing systems whose behaviour must adapt to their changing environment. To this end, the architecture implements a virtual machine operating environment based on Scheme and μClinux that encapsulates a CPU core, digital logic, generic I/O, network interfaces and domain-specific programming language composition.

1 Introduction

A sensor network can be viewed as a large-scale distributed system composed of diverse non-uniform hardware devices having both real-time performance and low-power design constraints. Applications running on such platforms must generally adapt their behaviour in response to user tasks, sensed information, dynamic changes in connection topology and temporary/permanent problems with the nodes and communications links present in the network. The adaption can range from simple adjustments of parameters through to partial or complete reprogramming of individual nodes (or indeed the entire network). Thus a sensor network can be viewed as a particularly demanding canonical example of a *context aware pervasive system* where context represents the dynamically-changing distributed environment from the point of view of nodes in the sensor network that are collectively executing one or more distributed applications.

Conventional techniques for the development of pervasive systems have focused either on event-based systems where behaviour is specified using processing tied to events, or on model-based systems using rules applied to a shared context model. [4] argues that both approaches suffer from fragmented application logic, and interactions between rules or events and processing must be analysed in conjunction with environment state information to determine if they result

in correct and stable behaviour. *Stability in the face of adaptation* is thus the major design challenge.

Emerging approaches to the development of pervasive systems utilise virtual machines and/or domain specific programming. Maté[2] is one such approach that has developed a virtual machine (VM) for nodes built directly on top of TinyOS[3]. We contend, however, that there may be considerable advantage in applying more advanced programming language approaches directly to sensor networks. Specifically we believe that allowing sensors to be programmed using their own domain-specific language constructs, taking advantage of innovations such as aspect-oriented programming and proof-carrying code, may make a major contribution towards the development of a stable, extensible, comprehensible context-aware systems consisting of thousands of elements.

The vision described in this paper is thus motivated both by developments in sensor hardware and platforms and by recent research in the semantics and construction of programming languages for context aware pervasive computing systems. We seek to combine the notion of context and dynamic domain specific languages into a single infrastructure, by providing:

- a single logical target architecture that can be applied to all nodes in a sensor network;
- an experimental hardware infrastructure based on μClinux[6]; and
- a reconfigurable programming platform using a Lisp- or Scheme[7]-like VM.

Our goal is to investigate language constructs for sensor networks while satisfying real-time performance and low-power design constraints. The dynamic domain-specific aspect we advocate differs from previous work in being based explicitly on a hardware/software co-design approach supporting the deployment of novel programming language constructs directly onto the hardware in order to improve optimisation and expressibility. This is significantly more extensible and portable than (for example) an implementation of Maté extended to dynamically load binary code.

Section 2 presents some basic requirements for hardware in pervasive computing systems in general and wireless sensor networks in particular, and then discusses dynamic domain specific languages from a pervasive systems perspective. Based on this, section 3 offers a research agenda for co-designed context-aware solutions, whose sensor network context is made concrete in section 4. Finally, section 5 concludes with some pointers to the future.

2 Requirements for truly pervasive computing

The development of a pervasive computing application has two logical focal points to its development: the local focus of a node and the collective focus of the network in achieving the objectives of the network application when environment and objectives are dynamically varying. Both focal points have hardware and software components that need to function synergistically, and so are perhaps best treated together.

2.1 Hardware requirements

Pervasive hardware suffers from a number of design constraints simply by virtue of being targeted at inconspicuous placement in the environment. Sensor networks highlight these constraints particularly clearly – although it is important to realise that they also apply more generally to systems that (for example) include handheld and other elements. A non-exhaustive list of requirements for such networks would include:

Self-organisation and adaptation A process must discover the availability and quality of network routes that change dynamically with environmental factors, mobility of nodes and temporary and permanent failures of nodes and communication channels. Desirable adaptation features include customisation of the communications protocol, medium access control and routing information. Adaptation is essential as the local and collective roles of the network and how data is processed and communicated are likely to be modified in response to changes in the environment, network applications in execution and tasks requested of applications.

Security mechanisms Unauthorised modification of network applications (especially for sensor networks) must be prevented. In many applications there may also be stronger privacy guarantees on the ability of outsiders to observe the data sensed or the population of the network.

Discovery Many networks are self-discovering and self-configuring, in the sense that there is no *a priori* communications or naming topology associated with the elements. The population of nodes can change dynamically over time[3], and applications must be able to tolerate (and preferably benefit from) this dynamism.

Power-aware Frequently there is no power distribution network physically connected to nodes and power is delivered using batteries and/or is scavenged from energy sources such as light, vibration, movement, stress or fluctuating magnetic fields. A key requirement is the ability to start and stop hardware services and to enter standby modes in order to reduce power consumption. This is of particular importance for any radio interfaces for network communication.

Synchronisation It is important to be able to synchronize time with groups of nodes, both for applications having fine-grained temporal context and to minimise power by ensuring that all nodes involved in communication during a particular finite time period have powered up and started their radio interfaces.

Each of these requirements consists of a hardware *and* a software component, with the latter itself consisting of knowledge representation and processing components. Power-aware systems, for example, have the following components:

[3] This is even true in *augmented materials* where elements are embedded at fabrication time. Element and communication failure make such materials dynamic, and it is often too complicated to pre-determine node locations and connections even given that they are embedded in a solid substrate.

Hardware The ability to logically start, stop, suspend and resume components, often in response to events.

Knowledge A model of the current context of operation and the set of active tasks in order to support decision-making and processing.

Processing Logic to decide which components may be stopped or suspended in a given situation.

The two software components might typically be fused together, but there are advantages to considering knowledge representation separately from processing. Equally there are co-design challenges in ensuring that the hardware provides the necessary features for software control, and that the software uses these features as well as possible.

2.2 Dynamic domain-specific languages

The evolution of a particular domain-specific programming language can be viewed as the search for the most appropriate mechanisms that express solutions to problems encountered by application developers in the domain. No one language can optimally represent all ways in which to solve a problem: consequently many different languages and techniques have evolved to address different application domains.

The significance of domain-specific languages is that they allow programmers to express directly the concerns of importantance to that domain. By making the concerns explicit, domain-specific languages can provide more structured information to compilers and other tools to inform optimisation.

One approach at unifying disparate languages is aspect-oriented programming[8] where a single language (or occasionally multiple languages[9]) is used to develop separately the individual concerns of a problem. The number and type of aspects are typically fixed at design time: they are then developed and tested separately prior to "weaving" the aspects together late in the development cycle. While aspect-oriented programming has had some successes, it cannot easily integrate new aspects dynamically into the language or program.

A complementary approach is to allow *languages themselves* to be constructed from smaller elements, allowing the construction of domain-specific systems via the composition of language component specifications[10]. A specification might include abstract syntax, concrete syntax, type rules, rewrite rules and perhaps supporting libraries. Libraries need implementations, but the other elements can be specified declaratively. A program can refer explicitly to the language components in which it was developed as part of its source code. A language is then defined by its components and associated evaluators[4] necessary for each of the components. An evaluator for a particular domain specific language is dynamically constructed by compiling the evaluators for its components[5]. A client that downloads a program implicitly discovers the programming language

[4] An evaluator can be an interpreter or a compiler but we will largely use the term compiler in the text.

at load time and need only create an evaluator for that particular language as and when execution of the program is required.

Both techniques rely on combining a collection of largely independent fragments in order to create a final program or language. To be useful, there must be both an identifiable set of fragments and a collection of implementations to provide a space within which different combinations may be tried.

3 Characteristic contributions from co-design to the research agenda for context-aware platforms

Pervasive and context-aware computing rely on the ability to "inject" sensing and computational intelligence into the wider environment, and so encourages the use of microsensing, *ad hoc* wireless networking and advanced reasoning techniques. There is an obvious tension between the requirements: it is relatively easy to construct microsensors and deploy them in a wireless network, but their small size and low power mitigates against including many of the advanced software techniques that are otherwise highly appropriate for managing the network and its results.

A pervasive computing network thus presents a programming platform operating under a unique set of constraints (section 2). It seems unlikely that programming languages evolved for different environments – desktops, servers, and even relatively high-power stand-alone handheld devices – will capture these constraints effectively. This is important both for systems designers (who may not get the best out of their systems) and developers (who will struggle with an inappropriate conceptual model and mode of expression).

However, the most important consideration comes from the ability to deploy sophisticated software anywhere in the environment. Pervasive computing in handicapped by being *asymmetrical*: most of the processing power resides in large dedicated computers. A good example is when assets are tagged with RF-ID tags: the infrastructure (typically a building) can "see" the asset tag, but the asset cannot respond to or make its own determinations about its environment. Constructing applications from networks of low-power nodes goes some way to restoring symmetry to the situation, in that they allow computing and sensing *within* (rather than simply *of*) the asset base. This is also important for scalability.

We contend that the way to address these issues is **to allow the language used in developing pervasive applications to be designed alongside the sensing infrastructure**. This does not preclude familiar constructs or re-use of ideas from other domains – actually quite the contrary – but suggests that **some novel forms will contribute strongly to the effective use** of such networks.

This tight coupling suggests that co-designed network elements may make some distinctive contributions to the research agenda in context-aware, self-deploying and self-managing pervasive computing. Without ignoring the other

issues of connectivity, protocols, security, discovery and so forth, here we concentrate on these novel contributions.

Desktop and server systems, despite differences in operating system and programming language, present an overwhelmingly homogeneous platform for developers – a considerable effort has been expended to make this so. Embedded networks are far more heterogeneous, and there is a danger that software will become too targeted at individual elements' capabilities to be able to deal with failure or relocation. We need to understand **what are the correct abstractions for targeting with heterogeneous networks efficiently without over-commitment** to particular details.

Pervasive – and especially sensor – networks can involve large numbers of elements (of the order of thousands). This is far larger than anything dealt with by all but a few distributed systems projects. We have little understanding of **how to efficiently distribute fine-grained functionality** in such systems. Such knowledge as does exist (largely from the high-performance computing communities) deals with static situations: **how we retain performance (in the widest sense) from fine-grained systems that need to constantly re-configure?** It is important to remember that "performance" needs to be understood broadly, as many pervasive computing systems may (for example) stress low power consumption at the expense of processing capacity.

Domain-specific languages suffer to some extent from an over-abundance of flexibility: developers require *some* stability, and one might argue that even a sub-optimal stable core is preferable to a system that is technically better but moving too fast to become expert at. Not all network and sensor features *need* language features: the question therefore arises as to **what is the correct methodology for determining the correct contributions of hardware and software within the co-design?**

It is also easy to forget that computers almost never run a single application, and this is unlikely to change for pervasive computing systems. The network will run several "applications" simultaneously, for different users and with differing (and possibly conflicting) abstractions of the outside world. **How should pervasive computing applications co-exist?** – both at the multiple semantic level as discussed in [4] and at the more prosaic level of allowing different applications, and possibly using completely different programming abstractions and languages. This is an attractive motivator for re-configuring the software capabilities of the network while retaining continuous service.

4 The sensor networks perspective

To make the above agenda more concrete, for the remainder of this paper we will focus on deploying our ideas in the context of wireless sensor networks. Our detailed architectural choices are conditioned by balancing the desire for an open, simple, extensible, rapid development platform against the desire for a solution that can be tested in realistic environments when appropriate. This

has led us to choose a hardware architecture based on Field-Programmable Gate Array (FPGA) technology, coupled with a highly dynamic software platform.

4.1 Hardware

The target architecture is based loosely on figure 1 and consists of power sources, FLASH, SRAM and possibly DRAM memory, CPU core, general purpose I/O, RF communication unit and digital logic with μClinux as the embedded operating system. We intend to prototype the architecture using Xilinx FPGA technology to implement the CPU core, digital logic and general purpose I/O. The CPU core is currently planned to use the OR1K open core because a stable μClinux port exists and the core has been successfully synthesised both in silicon (by Flextronics) and in FPGA devices. Although it may not be the most appropriate core for low-power sensor nodes, in theory the core can be used in all node classes in sensor network architectures. Techniques such as clock gating can be used to dynamically switch the processor and other functional units to low-power standby modes[5].

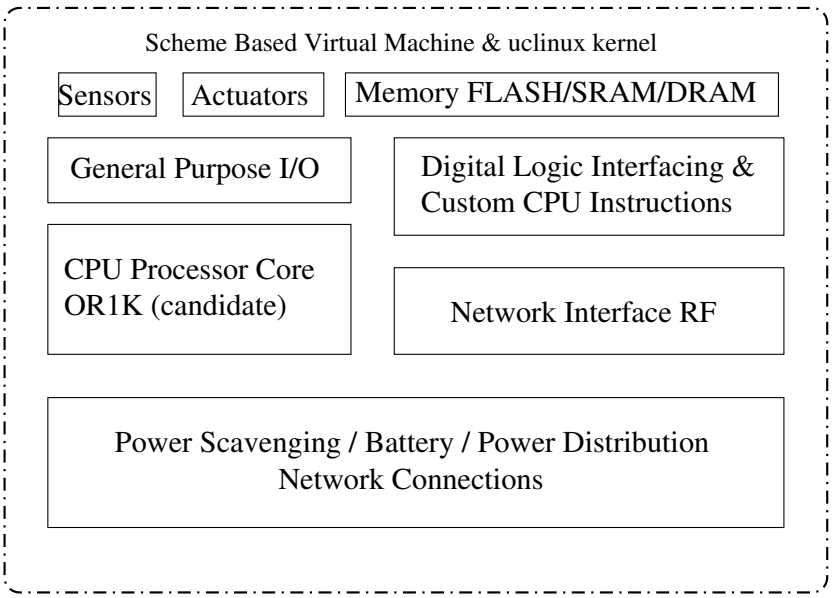

Fig. 1. Proposed target architecture.

[5] If the sensor node requires ultra low power operation in an energy scavenging environment then it is necessary to use aggressive techniques such as asynchronous logic to implement the architecture as a custom mixed-signal ASIC device.

The digital logic of the FPGA device can be used to interface to general purpose I/O, this will be necessary to connect sensors, actuators, memory and network interfaces (both RF and conventional) to the CPU. Obviously a physical digital interface to the RF will need to be presented to the general purpose I/O. Custom medium access controls can be implemented entirely in digital logic or with some software assistance. FLASH memory can be used to store FPGA configurations and boot images of uClinux and the Scheme based virtual machine operating system infrastructure. The digital logic can also be used to implement custom hardware accelerated user-instructions for the CPU core. Processing that does not map well to the CPU instruction set, or whose computationally requirements make it difficult to meet real-time performance constraints are candidates for implementation in hardware logic. Nodes requiring digital signal processing of audio/video data may require such functionality. The ability to add domain specific processing units in flexible digital hardware means that the target architecture should be capable of offering the performance necessary to implement high bandwidth sensors and gateway nodes. Additionally the use of our proposed architecture makes it possible to implement a systematic method for adding new sensing/actuating hardware that is accessible to our Scheme based software programming platform.

4.2 Software

We have chosen to base our programming platform on the Scheme language[7], for a number of reasons:

1. Scheme has clean semantics and concise syntax that can easily be supported on an embedded system;
2. it provides a rapid prototyping environment for sensor networks that can be easily simulated on desktop computers; and
3. it provides a scripting-based interface to programming sensor networks (as advocated as a useful feature in [11]) that will reduce the level of hardware knowledge required by users, without compromising the possibility of compilation and analysis.

We rejected the Java-based solution of [10] as too heavyweight for a large (and growing) number of sensor network applications, for which supporting a Java VM is either inappropriate or impractical. We rejected a C-based solution for reasons of complexity for application programmers.

4.3 Combination: μClinux and Scheme

A number of attempts[2, 13, 14] have been made to use high-level scripting languages and interpreters in order to simplify application development and to maintain code portability without sacrificing precise control over hardware. The problem then becomes how to maintain a high degree of efficiency alongside virtual access to hardware resources. We have chosen Scheme as the basis for our

scripting language (Common Lisp would also have been a valid choice, although more complex). The key question is: how do the Scheme virtual machine (VM) and μClinux interact? This determines to what degree we virtualise access to hardware and the overhead in using this abstraction.

The design options are for Scheme to be positioned:

1. Directly on top of the bare hardware. This would make it possible to construct the entire operating system in Scheme, and μClinux would not be required. Whilst this would be an interesting research direction it is likely to severly limit the space of designs and concepts that might be required as in Movitz[1] where Common Lisp was considered.
2. As a conventional program in a process like the shell. This would be similar in nature to running Scheme in a terminal window on a desktop computer. This approach is by far the easiest to implement but it is the least flexible in terms of operating system customisation. A suitable example of this is the Scheme shell scsh [15] which provides a scheme based scripting language and a posix interface with both high and low-level networking support.
3. Similar to the approach of Movitz, where Common Lisp is used to provide a framework for experimenting with kernel-level development programmed in Lisp. Movitz does not directly support threads and processes, nor does it have an implementation of SLEEP or contain assumptions about how to measure time.

In our work we envisage that the ideal place to position the Scheme VM lies somewhere between 2 and 3 but closer to 3.

There are essentially two choices for implementing a novel language on a sensor network. The traditional approach is to cross-compile the language from a standard desktop host, generating appropriate machine-language instructions. The generated code can be as efficient as handwritten code, although in practice it is typically significantly less so. However, cross-compilers are difficult to develop, debug and optimise.

The alternative is to provide a VM running on the sensor platform itself, accepting the performance and space penalties in order to improve flexibility and ease of development. This is practical only for very small VM run-time systems.

We are exploring both options, but tending towards the latter. Our reasoning is that – paradoxically – there are *fewer* power and space restrictions on a sensor network than in traditional distributed systems because of the sheer number of elements that can be deployed. The challenge is to provide a suitable distribution and co-ordination framework within which to deploy applications over a large number of elements. Using a VM on the elements allows us to focus on this challenge rather than on efficient cross-compilation.

We plan initially to implement 2 in order to provide a working experimental platform at the earliest possible opportunity whilst further researching an appropriate level for Scheme which will enable a sufficient degree of operating system customisation as part of the dynamic domain specific language definition. Clearly

our implementation must make modifications to the Scheme virtual machine in order to support the special requirements of sensor networks and dynamic domain specific languages.

4.4 Evaluation targets

Sensor networks may be used in a variety of real-world scenarios, ranging from earth science and monitoring to security and military applications. Any sensor network architecture must demonstrate an ability to target one or more of these application domains efficiently. The following applications illustrate three broad paradigms with which to evaluate our work, and show how our architecture can improve the ways in which they are addressed.

Habitat/environmental monitoring Nodes are used to sense features of the habitat that are of interest to scientists and environmental protection agencies over a period of months or even years. The network senses, processes and funnels data towards gateway nodes that are connected to the internet using standard protocols. The data may be pushed or pulled dependent on whether the sensor nodes or tasks (queries) from gateway nodes are the active entities initiating communication. A tree-based routing network must be constructed and maintained. Low-power operation is of prime importance for this application class but fine-grained synchronisation of nodes is usually of low importance. Whether the nodes are fixed or mobile is largely dependent on the habitat, for example sensor nodes circulating in a water system such as a lake will be mobile but nodes deployed by air drop onto a land mass are likely to be fixed.

Shooter localization The aim of the application is to determine the origin of a bullet or any other projectile in an urban environment. The nodes must sense the shock waves due to the projectile with a high sample rate and fine-grained time synchronisation in order to forward their data onto a gateway node and/or a server where the localisation is normally computed centrally. Power consumption will be significantly higher than in environmental monitoring.

Pursuer-evader/traffic management The aim of this application is to track the movement of one or more evader robots. The network must route this information to one or more pursuer robots using a routing protocol that exploits knowledge of geographic position information. An obvious extension of this application paradigm would be the more general problem of traffic management where sensors are present in cars, traffic lights and CCTV and speed cameras. The goals and tasks of this traffic application encompass congestion reduction, enforcing road/driving regulations for safety and informing law enforcement and accident and emergency services of appropriate events requiring their intervention.

In each case there are clear hardware and software constraints that must be met by any proposed solution. Our approach is to address these constraints

via co-design, ensuring that the appropriate language constructs are backed-up by appropriate hardware capabilities. Engineering such solutions pose an interesting challenge: how does one determine the success of a language construct, especially in conjunction with a hardware platform? There is little clear existing engineering methodology to apply to this problem.

5 Conclusion

We have motivated and presented the design of a new architecture for the nodes of a sensor network. The architecture differs from previous work in being based explicitly on a hardware/software co-design approach supporting the deployment of novel programming language constructs directly onto the hardware in order to improve optimisation and expressibility.

Although we have stressed the co-design aspects with reference to small devices, the software techniques can be applied to more traditional platforms as well. This means that a similar domain-specific language could be used across a range of scales, with (for example) some language features being (de)selected on some platforms.

We are currently completing feasibility studies on the components of our proposed architecture, prior to initial development work. Our immediate research challenges are to determine appropriate abstractions for the construction and deployment of the embedded systems architecture from hardware and software perspectives. We intend to evaluate our work against a range of applications, both to check the qualities of individual solutions and to derive methodological understanding that aids the creation of complex co-designed sensor networks.

References

1. Fjeld, F.V.: Movitz: Using Common Lisp for kernel-level programming on commodity hardware. *Workshop on Evolution and Reuse of Language Specifications for Domain Specific Languages at ECOOP2004*.
2. Levis, P., Culler, D.: Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, pages 85-95, San Jose, California, October 2002.
3. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for network sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Cambridge, USA, November 2000.
4. Dobson, S., Nixon, P.: More principled design of pervasive computing systems. In *Proceedings of Engineering for Human-Computer Interaction and Design, Specification and Verification of Interactive Systems (EHCI-DVIS04)*, To appear in LNCS.
5. Dobson, S.: Creating programming languages for (and from) the internet. *Workshop on Evolution and Reuse of Language Specifications for Domain Specific Languages at ECOOP2004*.
6. uClinux Embedded Linux Microcontroller Project Home Page: http://www.uclinux.org/

7. Lord, T.: Guile Scheme, Technical Report (1996), online 'info' documentation, Free Software Foundation.
8. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings of the European Conference on Object-Oriented Programming. Volume 1241 of LNCS. Springer-Verlag (1997) 220–242
9. Lafferty, D., Cahill, V.: Language-independent aspect-oriented programming. In: Proceedings of the ACM Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA'03), ACM Press (2003)
10. Dobson, S., Nixon, P., Wade, V., Terzis, S., Fuller, J.: Vanilla: an open language framework. In Czarnecki, K., Eisenecker, U., eds.: Generative and component-based software engineering. LNCS. Springer-Verlag (1999)
11. Hill, J. Horton, M., Kling, R., Krishnamurthy: The Platforms Enabling Wireless Sensor Networks. Communications of the ACM, Volume 47, number 6, pages 41-46, June 2004.
12. Levis, P., Madden, S., Gay, D., Polastre, J., Szewczyk, Woo, L., Brewer, E., Culler, D.: The Emergence of Networking Abstractions and Techniques in TinyOS. *Proceedings of the First USENIX/ACM Symposium on Networked System Design and Implementation, NSDI 2004.*
13. Girod, L., Elson, J., Cerpa, A., Stathopoulus. T., Ramanathan, N, Estrin, D.: Em*: A Software Environment for Developing and Deploying Wireless Sensor Networks, Technical Report 034, Centre for Embedded Network Sensing, UCLA Computer Science Department, Los Angeles, USA, 2003.
14. Madden, S., Szewczyk, R., Franklin, M., Culler, D.: Supporting Aggregate Queries over Ad-Hoc Wireless Sensor Newtworks. *Proceedings of the Workshop on Mobile Computing and Systems Applications*, New York, USA, June 2002.
15. Scsh - The Scheme Shell Home Page: http://www.scsh.net/