

**C@t: A Language for Programming
Massively Distributed Embedded Systems**

by

Devasenapathi P. Seetharamakrishnan

Bachelor of Engineering in Electronics & Communication
Government College of Technology, Coimbatore
June 1991

Submitted to the Program in Media Arts & Sciences,
School of Architecture and Planning
in partial fulfillment of the requirements for the degree of
Master of Science in Media Technology

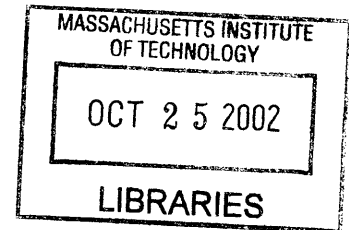
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

© 2002 Massachusetts Institute of Technology.
All rights reserved.

ROTCH



Author
Program in Media Arts & Sciences
August 31, 2002

Certified by
V. Michael Bove, Jr.
Principal Research Scientist, Media Arts and Sciences
Thesis Supervisor

Accepted by
Andrew B. Lippman
Chairman, Department Committee on Graduate Students

C@t: A Language for Programming Massively Distributed Embedded Systems

by

Devasenapathi P. Seetharamakrishnan

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning
on August 31, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Media Technology

Abstract

This thesis presents **c@t**, a language for programming distributed embedded systems that are composed of thousands (even millions) of interacting computing devices.

Due to the improvements in fabricating technologies, it is becoming possible to build tiny single-chip devices equipped with logic circuits, sensors, actuators and communication components. A large number of these devices can be networked together to build Massively Distributed Embedded Systems (MDES). A wide variety of embedded control applications are envisioned for MDES: responsive environments, smart buildings, wildlife monitoring, precision agriculture, inventory tracking, etc.

These examples are compelling, however, developing applications for MDES remains complex due to the following issues: MDES consist of large number of resource constrained devices and the number of potential interactions between them can be combinatorially explosive.

Systems with the combined issues of such scale complexity, interaction complexity and resource constraints are unprecedented and cannot be programmed using conventional technologies. Accordingly, this thesis presents **c@t**, a language that employs the following techniques to address the issues of MDES: 1. To address the scale complexity, **c@t** provides tools for programming the system as a unit. 2. **c@t** offers a declarative style network programming interface so that network interactions can be implemented without writing any low-level networking code. 3. The applications developed using **c@t** are vertically integrated. That is, the compiler customizes the runtime environment to the suit the application needs. Using this integrated approach, efficient applications can be developed to fit the available resources.

This thesis describes the design, features and implementation of **c@t** in detail. A sample application developed using **c@t** is also presented.

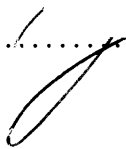
Thesis Supervisor: V. Michael Bove, Jr.
Principal Research Scientist, Media Arts and Sciences

C@t: A Language for Programming Massively Distributed Embedded Systems

by

Devasenapathi P. Seetharamakrishnan

Thesis Reader

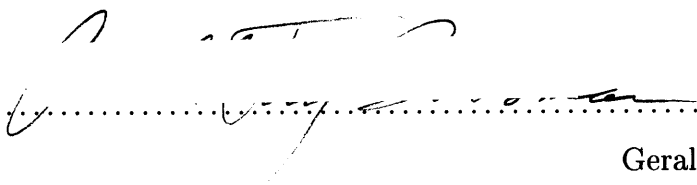


Joseph A. Paradiso

Associate Professor of Media Arts and Sciences

MIT Media Laboratory

Thesis Reader



Gerald Jay Sussman

Matsushita Professor of Electrical Engineering

MIT Department of Electrical Engineering and Computer Science

Acknowledgments

First and foremost my thanks to my erstwhile advisor H. Shrikumar. He inspired me to do good research and supported me and my work with sincerity and clarity.

Prof. Mike Bove kindly adopted me as his student, after Shrikumar's departure. I am grateful to him for his kindness, advice and support.

To my readers, Prof. Joe Paradiso and Prof. Gerry Sussman, I offer my sincere thanks; you have done me a great honor. I am very grateful to Prof. Gerry Sussman for valuable pointers and insightful advice. He made time for (even encouraged) a number of meetings to discuss the issues and crystallize the ideas.

My special thanks to Prof. Neil Gershenfeld for admitting me to the Physics and Media Group. I am indebted to Prof. Whitman Richards for inspiring me to see the regularities in myriad details.

Bill Butera was my zen master. I am indebted to him for providing me intellectual and emotional support.

Stefan Marti was my unofficial "big brother". He has helped me in many possible ways - making posters, reviewing my drafts, teaching the ways of the lab, chocolates,.. an endless list.

Josh Lifton and Radhika Nagpal offered immeasurable help and insight.

I am also grateful to many students for their help and support. Thanks to Marco Escobedo, Rich Fletcher, Ashish Kapoor, Natalia Marmasse, James McBride, TJ Mcleish, Surj Patel, Ali Rehim, Ramesh Srinivasan, Sunil Vemuri, Ben Vigoda, and Jim Youll.

I was also fortunate to receive valuable hints and notes from researchers through newsgroups and mailing lists. Thanks to Prof. Steven Johnson of Indiana University and Dr. Aubrey Jaffer of the scheme community.

This thesis is developed entirely using the software that is freely available. I thank the following organizations for distributing valuable software resources freely: GNU (Debian Linux, XEmacs, GCC, Latex, Simulpic, etc), Hi-Tech (PICC Lite compiler),

Javasoftware (Java 2), and PLT Schemers (MZScheme).

Linda Peterson safely carried me through various obstacles. I can't thank her enough.

I am thankful to Satish Raj, my friend, philosopher and guide for inspiring me to seek quality in everything I do.

Finally, thanks and love to my family and in particular to my wife, Sonal Shastri for her unfaltering love and support.

Contents

- 1 Introduction** **10**
- 1.1 System Characteristics and Issues 10
 - 1.1.1 Scale Complexity 12
 - 1.1.2 Interaction Complexity 12
 - 1.1.3 Spatial Relations Between Devices 13
 - 1.1.4 Minimal Resources 13
- 1.2 c@t Overview 14
 - 1.2.1 Collective Programming 15
 - 1.2.2 Associative Naming Scheme (ANS) 17
 - 1.2.3 Declarative Network Programming 18
- 1.3 Outline of Thesis 19

- 2 Related Work** **20**
- 2.1 Programming Models 20
 - 2.1.1 Amorphous Computing 21
 - 2.1.2 Paintable Computing 22
- 2.2 Programming Languages 22
- 2.3 Network Architectures 23

- 3 The c@t Language** **25**
- 3.1 Lexical Conventions 26
 - 3.1.1 Identifiers 26
 - 3.1.2 Whitespace and Comments 26
 - 3.1.3 Other Notations 27
- 3.2 Basic Concepts 27
 - 3.2.1 Variables 27
 - 3.2.2 Regions 27

3.3	Compiler Directives	28
3.3.1	Device Declarations	29
3.3.2	Device Set Specifications (DSS)	29
3.4	Expressions	31
3.4.1	Primitive Expression Types	31
3.4.2	Derived Expression Types	35
3.5	Support for Low-level Programming	37
3.5.1	Special Function Registers and Ports	37
3.5.2	Including Assembly Language Code	38
3.5.3	Interrupt Handling	38
4	Runtime Environment	39
4.1	Runtime Design	40
4.1.1	Remote Symbol Reference Subsystem	40
4.1.2	Message Interpreter	44
4.2	Runtime Implementation	45
4.2.1	Remote Symbol Reference Subsystem	45
4.2.2	Message Interpreter	46
5	Application Example	49
5.1	Simulation Environment	49
5.2	Get-Set-Go: A Self-organizing Building Control System	51
6	Conclusions and Future Work	55
A	Formal Syntax	59
B	Source Code	64
	References	67

List of Figures

1-1	Sample Application.	14
1-2	Functioning of the c@t Compiler.	16
3-1	Program Regions.	28
4-1	The Code Components Generated by the c@t Compiler.	39
4-2	The RSR Components and Their Interactions for a Simple Call.	42
4-3	The Sequence of Events in Invoking a Remote Function.	43
5-1	Simulator Architecture.	50
5-2	Simulator Screen Shot.	52
6-1	An Embedded Device Network.	56

List of Tables

2.1	Comparisons with Parallel and Distributed Programming Languages.	24
3.1	Metacharacters of Count Expression.	31
4.1	A Message Example.	46
4.2	The Message Structure in the Runtime Environment for Microchip PIC 16F84.	48
5.1	Memory Usage Statistics.	53

Chapter 1

Introduction

DUE to advances in fabrication technologies, it is becoming possible to fabricate single-chip devices that contain logic circuits, sensors, actuators and communication components. These devices are simple and tiny, and yet they can be networked together to build Massively Distributed Embedded Systems (abbreviated as MDES). These systems can enable a wide variety of distributed embedded control systems: responsive environments, environmental monitoring, inventory tracking, precision agriculture, wildlife tracking, etc.

These applications are compelling, but, due to some unique issues of MDES, there are several challenges to programming and networking them. This thesis presents a language, named `c@t`¹, that attempts to address these challenges and to facilitate programming MDES.

This chapter discusses the challenges of MDES and presents a conceptual overview of `c@t`.

1.1 System Characteristics and Issues

The unique characteristics of MDES are illustrated through a sample application - distributed building networks.

Researchers [18] are developing concepts for creating reconfigurable buildings from an integrated chassis that can be rapidly and precisely installed with minimal field la-

¹The name is derived from Computation at a point in space (@) and Time.

bor. In one integrated assembly, pultrusion glass fiber composite beams and columns provide structure, insulation, sensor arrays, lighting, signal and power cable raceways, and ductwork. The chassis provides the necessary physical, power, and signal connections for mass customized infill components to be quickly installed, replaced and upgraded without disruption. Infill components may include integrated wall/floor assemblies, specialty millwork with transformable elements, display systems, networked appliances and devices, etc.

Dynamically reconfigurable buildings can lead to substantial reductions in building costs and to efficient space utilizations.

However, if buildings are to be reconfigurable, the electric networks must be reconfigurable too. That is, it must be possible to modify electrical networks according to the changes in building/room structures. For instance, if a single big room R_0 is divided into two small rooms R_1 and R_2 , the electrical appliances have to behave accordingly; a switch in R_1 should control only the lights in R_1 and not the ones in R_2 (even though that might have been correct in the old configuration).

A novel building network [27] to support reconfigurable buildings is being developed. In this network, associations are in software instead of the usual hardware connections. That is, every electrical component is connected (directly or indirectly) to every other component and the semantic associations are created and deleted if when and necessary.

Every electrical component, beam, column and infill panel is equipped with embedded computing devices. These computing devices self-configure into electrical networks that reflect the building structure.

The unique characteristics of MDES that can be observed from this example are:

- Scale complexity - Extremely large number of devices.
- Interaction complexity - Potentially combinatorially explosive number of interactions between those devices.
- Spatial relations - Spatial and structural relations between devices form a significant part of the computations.

- Resource constraints - Devices are usually designed with minimal resources to reduce cost and size of devices.

These characteristics pose some unique challenges and issues and they are discussed next in detail.

1.1.1 Scale Complexity

MDES consist of extremely large number of nodes. For instance, a building network that supports a large building would be composed of hundreds of thousands of embedded devices. In fact, building control systems with tens of thousands of nodes already exist [12]. Shrikumar [28] and Takada et al [30] predict that the next wave of distributed embedded systems will be composed of thousands to billions of tiny devices.

Conventional distributed computing technologies involve several manual operations - configurations, topology design, maintenance, etc. As D.Tennenhouse [31] remarks, these technologies are not suitable for MDES, as the ratio of number of devices to number of humans would be too large for human-centric solutions.

1.1.2 Interaction Complexity

In MDES, it would be necessary for many devices to cooperate to complete an application task. The computations performed by individual devices might be simple but the tasks completed collectively could be substantial. For example, in building networks, neither temperature sensors nor fan controllers can complete any significant application tasks on their own. But, by cooperating, they can maintain the desired temperatures in the building.

Coordination and cooperation between a large number of devices would require a large number of network communications. This is unlike conventional computers where individual nodes perform sizeable tasks and coordinate with other computers when there is a need.

Moreover, since MDES might be deployed in unconventional environments, they may not have any network infrastructure elements such as nameservers and routers.

As a result, the devices would need to perform both the application and the networking tasks such as routing, naming services, etc.

Given the large number of interactions and the lack of network infrastructure, developing distributed applications for MDES using conventional programming languages (usually, embedded applications are written in C, Forth, Assembly, Java Microedition, etc) would be complex and tedious.

1.1.3 Spatial Relations Between Devices

Embedded devices are tightly coupled to the physical world and the principal role of embedded software is not the transformation of data, but rather the interaction with the physical world. As a result, the spatial relations between devices becomes an important factor. For example, in building networks, embedded computing devices need to self-configure into building control systems. This requires that the devices will be able to determine the room they are located in, compute the shape of the room they are in, the wall they are attached to, etc; whereas in current distributed computing technologies, geometric and spatial relations are considered to be irrelevant.

1.1.4 Minimal Resources

Individual devices are deliberately designed with minimal computational resources, such as memory, processing power and communication capacity. There are two reasons for minimizing the resources:

- **Cost Reduction** - Should computers be truly ubiquitous, they must cost almost nothing. As N. Gershenfeld states [14], a business card that can call up a business's web page would be convenient, but there wouldn't be a business left if its cards cost more than a few cents .
- **Size Reduction** - Devices must be vanishingly small to be literally embedded in objects. If they are tiny, they will not only occupy less space (and volume), but they will weave themselves into the fabric of everyday life. Such a disappearance is a fundamental consequence not of technology, but of human psychology [33].

Since these devices have limited resources, both communications and computations must be efficient. That is, the code must minimize message complexity, time complexity and space complexity. Producing such efficient code using conventional languages is difficult, if not impossible.

1.2 c@t Overview

This thesis introduces c@t, a language that attempts to address the above mentioned challenges and to facilitate developing applications for MDES. This section presents the fundamental ideas and techniques of this language.

Figure 1-1 gives a simple temperature control system written in c@t. In this application, temperature sensors monitor the ambient temperature and if the temperature is greater than or equal to seventy degrees, they invoke the function `activate` on all the fans that have more than 0.5 units of battery power.

```
(declare-device sensor ((processor ``16F628``))
(declare-device fan ((processor ``18F2320``))
(declare-cluster temp-control ((sensor 160) (fan 90)))
(define (@ (= device sensor)...) float temperature 0)
(define (@ (= device sensor)...) void (monitor)
  (if (>= temperature 70)
    (activate (@ (grammar relational)
      (filter (and (= type fan) (> battery 0.5) ))))))
(define (@ (= device fan)...) void (activate)
  (set! RB7 #x80))
```

Figure 1-1: Sample Application.

A c@t program consists of four parts:

1. Device Declarations - Specifies the devices that are part of the system. The temperature control system consists of two types of devices - fans and sensors.
2. Cluster Declarations - Specifies the cardinality of every type of device in the system. In this example, there is one cluster called `temp-control` and it is comprised of 160 sensors and 90 fan controllers.

3. Device Set Specifications - An embedded language² that can be used to select a subset of devices from the set of declared devices. Device set expressions start with the `@` operator. For example, the variable `temperature` is defined on sensors using the expression `(@ (= device sensor))`.
4. Variables and Functions - As in the other high-level languages, programs can be composed using functions and variables. However, they can be defined on and referenced from multiple devices using device set specifications. For example, the function `activate` that is defined on fans is invoked from `monitor`, a function defined on sensors.

`c@t` employs the following techniques to enable programming MDES easily and efficiently:

- Collective programming - Multiple devices can be programmed together “*System as a unit*” approach. The language allows the programmers to view and program the system as a whole without worrying about the myriad devices and details.
- Associative Naming - Devices can be addressed based on intentions, functions, states and roles rather than just upon the numeric identifiers.
- Declarative network programming - Interactions between devices can be implemented without writing low-level network code.

In the next few sections, these techniques are explained in detail.

1.2.1 Collective Programming

Scale complexity of MDES is alleviated using the collective programming approach. In `c@t`, users program just a *small number of virtual components* which get automatically realized into potentially *a much larger number of physical components*. This is possible because MDES are usually composed of a small number of equivalent classes

²It is similar to how regular expressions are embedded in languages such as Perl and Awk.

of devices. The equivalence could be in terms of the functions performed or their properties or their current states. For example, although the temperature control application consists of 250 individual devices, they can be classified into sensors and fans based on their roles.

In c@t, devices can also be classified using their dynamic characteristics such as current states. For instance, the function `active` is invoked on fans, but only on those fans that have more than 0.5 units of battery power.

The Figure 1-2 shows the functioning of the c@t compiler. It takes a single sequential c@t program that describes the system level behavior and produces code files for each target device that is part of the system. For this temperature control application, the compiler would produce 250 code files that would execute on 160 sensors and 90 fans.

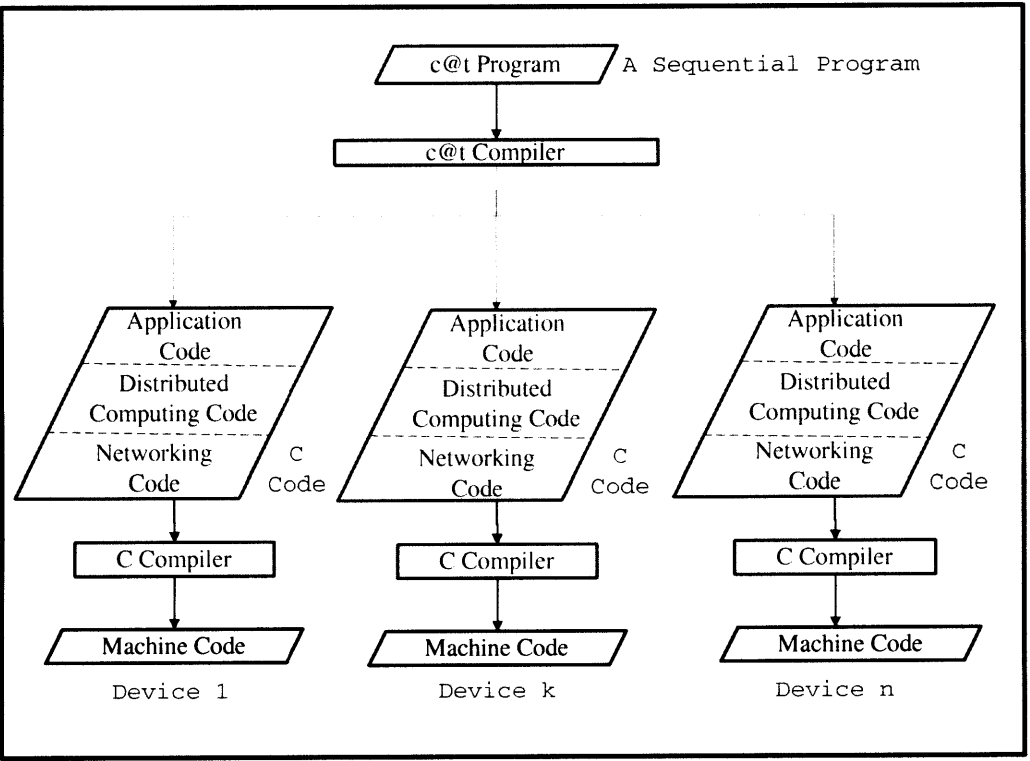


Figure 1-2: Functioning of the c@t Compiler.

1.2.2 Associative Naming Scheme (ANS)

Since the systems would be composed of equivalent sets of devices, communication would be not between individual devices, but between sets of them. It is important to note that some form of identification is necessary to distinguish between members of a set of equivalent devices. For example, to communicate with specific sensors, the individual devices in a sensor network need to be uniquely identified. Unlike with conventional networks, this identifier can be derived from the device attributes. For instance, two different sensors could be distinguished based on their physical locations.

To specify communications between sets of devices, a novel naming scheme called Associative Naming Scheme (ANS) has been developed. ANS is a naming mechanism that can be used to name devices based on their static characteristics (type, role, position etc) or their dynamic characteristics (current state, variable value etc).

Different generative languages are suitable for expressing different types of classifications. For instance, relational expressions can be used to select devices based on their attributes. Spatial formalisms such as geometrical equations, Lindenmayer systems [22] and fractals [23] can be used to select devices based on their spatial and structural properties. For example, in a wildlife tracking sensor network, a motion sensor, might need to coordinate with other sensors in a circular space to determine the direction and speed of animals. The area could be easily expressed using the geometrical equation of the circle.

The grammar field in the specification is used to choose between different grammars. For example, the expression (`@ (grammar relational) (filter (and (= device fan) (< hop 3))))`) uses a relational expression to choose devices. In the current implementation, only relational expressions can be used for specifying device sets.

The ANS is inspired by the the Intentional Naming System (INS) [3], a resource discovery and service location system for dynamic and mobile networks of devices and computers. ANS is more flexible as different types of grammars and the device count can be used in selecting target devices.

1.2.3 Declarative Network Programming

In c@t, the interactions between devices can be implemented without writing any low-level networking code. c@t uses the paradigms of function calls and variable references to represent the interactions between devices. Further more, these transfers of control and data can be implemented without writing any low-level networking code. For instance, the function `activate` defined on fan controllers is invoked by the function `monitor` defined on sensors as seamlessly as invoking a local function.

The machine code produced by the c@t compiler is vertically integrated. That is, the c@t compiler not only produces the application code, but also every single code component that runs on devices. This integrated approach can lead to efficient code realizations, as all the components can be tailored to the needs and characteristics of the application. For instance, the temperature control system can be realized in at least three of many possible ways:

1. A centralized solution, where a powerful device (if available) is chosen as a registry and all the other devices register themselves with that registry. When the sensors need to activate a fan controller, they can search this registry to choose an appropriate device and send an activation message.
2. A completely decentralized solution, where devices form minimum spanning trees to communicate and interact. Every time there is a temperature change, the sensors could search the neighborhood for fans and notify the best one. This solution is best suited for situations where there is no powerful device to act as a central registry and the searching for devices would not be expensive.
3. A hybrid solution, where many clusters are formed with one fan, one heater and multiple sensors. This solution assumes that fans can service multiple sensors simultaneously.

A solution can be selected based upon the required performance (limit on number of messages exchanged, reliability, duration of operation etc) and the available resources (computational resources, interconnection topology etc). Such extensive

analysis is beyond the scope of current implementation. Currently, this work focuses on language design, development of the c@t compiler, and the implementation of a runtime environment.

1.3 Outline of Thesis

The rest of thesis is organized as follows. Chapter 2 reviews prior work related to MDES and c@t. Chapter 3 outlines the basic concepts of the language, then provides a detailed description of the language constructs. Chapter 4 describes the design of the runtime environment and its implementation for Microchip midrange microcontrollers. Chapter 5 presents our experience developing a simple distributed embedded system using c@t. Chapter 6 discusses several directions for future research and presents the conclusions.

Chapter 2

Related Work

THE issues of MDES are not unique in and of themselves. Other areas of computing have worked out solutions for many similar issues. In fact, this work is inspired by several engineering techniques and tools: Associate Naming Scheme is inspired by Intentional Naming System [3]; the idea of synthesizing applications from minimally constrained programs using extensive analysis is borrowed from VHDL [13]; and, the elements of Device Set Specifications are loosely based on the Perl regular expressions.

Although some of the issues of MDES are handled by conventional technologies, the combination of these issues is unprecedented. Existing technologies don't work for MDES as they make several assumptions - amongst them: that there are central coordinating entities, that the ratio of number of computers to number of humans is small, that the computational devices are unique, etc - that are not true for MDES.

This chapter discusses the research works related to this thesis. These related efforts can be broadly classified into three main topics: programming models, programming languages, and network architectures for embedded devices. Each of these topics are discussed in detail below.

2.1 Programming Models

Researchers have proposed two programming models - Amorphous computing [1] and Paintable computing [8] for massively distributed embedded systems (MDES).

2.1.1 Amorphous Computing

The amorphous computing model was developed for programming amorphous computers. An amorphous computer consists of myriad computing particles embedded in physical space. The particles have limited computational resources; they run asynchronously, communicate with each other over a very limited physical distance, and are placed arbitrarily with respect to each other.

Two application specific programming languages were developed for this computing model: 1. Growing Point Language (GPL) [11], and 2. Origami Shape Language (OSL) [25].

Growing Point Language (GPL)

D. Coore has developed GPL using the botanical metaphor of “growing points”. A growing point is a locus of activity in an amorphous computing medium. It can be propagated to an overlapping neighborhood. Growing points can split, die off, or merge with other growing points. As a growing point passes through the medium, it affects the differentiation of the behaviors of the computing elements it visits. The growing point may be sensitive to particular diffused messages, and in propagating itself, it may exhibit a tropism toward a source, away from a source or move in other ways based on concentrations of diffused messages. In this way, GPL can generate various patterns such as Euclidean constructions, the interconnect topologies of an electronic circuit, etc.

Origami Shape Language (OSL)

In her Ph.D thesis, R. Nagpal [25] presents OSL, another language for programming amorphous computing devices. OSL is a language for instructing a sheet of identically-programmed, flexible, autonomous agents (“cells”) to assemble themselves into a predetermined global shape, using local interactions. With this language, a wide variety of global shapes and patterns can be described at an abstract level, compiled into cell programs, and then synthesized using only local interactions between identically programmed cells. Examples include flat layered shapes, all plane Euclidean

constructions, and a variety of tessellation patterns.

Both these languages focus on generating complex collective patterns from the local interactions of individual devices. They are not geared for general purpose programming.

2.1.2 Paintable Computing

W. Butera [8] developed paintable computing and the corresponding programming model. A paintable computer is defined as an agglomerate of numerous, finely dispersed, ultra-miniaturized computing particles; each positioned randomly, running asynchronously and communicating locally. The programming model is based on mobile processes and environmental support for the process mobility, scheduling and data exchange.

Although the paintable computing model is an interesting way to program MDES, no programming language has been developed for implementing this model. For instance, J. Lifton [21] has implemented this programming model in C on Pushpin Computers.

2.2 Programming Languages

The area of distributed embedded systems is actually a specialization of both distributed computing and parallel computing systems. Due to the common characteristics, several useful techniques can be borrowed from distributed [19, 6] and parallel computing languages [20]. Despite the similarities, none of these languages have all the features that are necessary to address the challenges discussed in Section 1.1. Table 2.1 compares c@t, and parallel and distributed programming languages.

Technically, many languages such as C, Java and processor specific assembly languages can be used to program MDES. However, the task becomes tedious as these languages don't have the right tools, abstractions, and constructs. As Abelson et al [2] remark, a programming language is more than just a means for instructing a computer to perform tasks. The language also serves as a framework with which we organize our ideas about systems. That is why some languages with the right set of

abstractions and expressions are more suitable for developing some types of systems than others.

2.3 Network Architectures

Sun Microsystems Inc. has developed JINI [32] to facilitate programming distributed embedded systems. JINI is a set of Java APIs and network protocols that can be used to build and deploy distributed systems that are organized as set of services. A service is any useful function offered by the devices in the network. For instance, a JINI-enabled printer could offer a printing service.

JINI defines a runtime environment that provides mechanisms for adding, removing, locating and accessing services. The devices that provide services add themselves to service registries. Clients locate these services by querying these registries. Once they find the required services, the clients can invoke the appropriate methods on the service provider objects to avail their services.

JINI imposes a centralized architecture on the applications. Centralized solutions have a few disadvantages:

- The registries have to be powerful enough to coordinate a large number of devices. Further, those nodes may be single points of failure.
- More importantly, the registries have to be carefully positioned at the appropriate locations to serve the other nodes. That is, if registries are separated from the other devices, just the management messages such as service discovery and service registration would far outnumber the application messages.

In c@t, the programs are not bound to any architectures. Since the applications are vertically integrated, the compiler can produce the most effective implementation of the application code.

Feature	c@t	Parallel Languages	Distributed Languages
<i>Example Languages</i>	-	Concurrent C, Oc- cam, StarLisp, etc	SR, Ada, Erlang, NIL, etc
<i>Primary Objective</i>	A high-level language that can be used to collectively program a very large number of devices.	These languages aim to exploit the parallelism of multiprocessor systems to achieve maximum performance and reduce computing time.	These languages aim to provide support for concurrency, communication and failure detection to build applications using multiple computers.
<i>Primary Target Architectures</i>	Systems with a very large number of tiny computing devices; e.g. sensor networks and distributed control system	Multiprocessor systems with a number of powerful processors; e.g. CM-2 of Thinking Machines and MP-2 of MasPar.	Multicomputer systems; e.g. telephone switches, replicated database servers etc.
<i>Collective Programming</i>	Yes, a large number of devices can be collectively programmed using predicate and pattern languages	Yes, multiprocessor systems can be programmed using implicit and explicit parallel programming languages.	Usually not possible, as there is not much equivalence between nodes.
<i>Spatial relations between nodes</i>	Can be specified using various pattern languages	Usually considered to be irrelevant	Usually considered to be irrelevant
<i>Networking</i>	Integrated network programming and intentional communications	Not applicable	Layered networking and usually ID based communications.

Table 2.1: Comparisons with Parallel and Distributed Programming Languages.

Chapter 3

The c@t Language

The goal of c@t is to help manage the scale and interaction complexities of MDES by preserving the sequential programming paradigms for distributed computations. c@t satisfies this goal by using the techniques of collective programming and declarative network programming.

The design of c@t language is heavily influenced by Scheme. c@t, like Scheme, employs a fully parenthesized prefix notation for programs and data. This language is statically scoped with a block structure established by the enclosing functions.

c@t is a statically typed or a strongly typed language. Types are associated with variables. Every program statement is checked for the correct type usages and promotions.

Arguments to c@t procedures are always passed by value, which means that the actual argument expressions are evaluated before the procedure gains control, whether the procedure needs the result of the evaluation or not.

The c@t compiler is written in Scheme (PLT Scheme [26]). As shown in Figure 1-2, the c@t compiler translates the programs to ANSI C and employs a C compiler to generate processor-specific machine code. The current implementation uses the Hi-Tech PICC Lite compiler [16] to generate code for midrange PIC processors.

The advantages of the translating to C are:

- The c@t compiler can utilize many of the low-level service routines provided by the C compiler. For instance, the PICC lite compiler provides memory initialization routines, interrupt handling routines, power on reset code, floating

point routines etc.

- Since `c@t` translates programs to ANSI C and C compilers are available for many commercially available processors, it becomes easy to support a wide range of processors immediately without writing any processor-specific code generators.

This approach has one disadvantage: since the `c@t` compiler doesn't have complete information about the low-level code generated by the C compiler, it could be a barrier to producing integrated applications. However, the `c@t` compiler can reasonably approximate the necessary information and still function adequately.

This chapter presents the concrete syntax and semantics of expressions, programs, and definitions. The formal syntax is presented in Appendix A.

3.1 Lexical Conventions

This section presents the lexical conventions used in writing `c@t` programs. Upper and lower case forms of a letter are distinct both within character and string constants, and the program elements.

3.1.1 Identifiers

An identifier is an unlimited-length sequence of alphabetic characters, underscore and digits, the first of which must be an alphabetic character. An identifier cannot have the same spelling as a keyword.

3.1.2 Whitespace and Comments

Whitespace characters are spaces and newlines. Whitespace is used for improved readability and as necessary to separate tokens from each other, a token being an indivisible lexical unit such as an identifier or number, but is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur inside a string, where it is significant.

A semicolon(`:`) indicates the beginning of a comment. The comment continues till a newline is encountered.

3.1.3 Other Notations

The following notations are important:

- `()` - Parentheses are used for grouping and to notate lists.
- `'` - The single quote character is used to indicate constant data.
- `"` - The double quote character is used to delimit strings.
- `\` - Backslash is used in the syntax for character constants and an escape within string constants.

3.2 Basic Concepts

3.2.1 Variables

An identifier that names a location where a value can be stored is called a variable and is bound to that location. The set of all visible bindings in effect at some point in a program is known as the environment in effect at that point. The value stored in the location to which a variable is bound is called the variable's value.

3.2.2 Regions

Like Scheme, `c@t` is a statically scoped language with block structure. To each place where an identifier is bound in a program, corresponds to a region of the program text within which the binding is visible.

The region is determined by the particular binding construct that establishes the binding. For example, if the binding is established by a function definition, then its region is the function definition. Every mention of an identifier refers to the binding of the identifier that established the innermost of the regions containing the use, then the use refers to the binding for the variable in the top level environment, if any; if there is no binding for the identifier, it would lead to a compilation error.

As the compiler translates the programs to C and since C doesn't supported nested function definitions and variable definitions within expressions, the regions established

by functions can be deepest in the hierarchy of regions. That is, identifiers cannot be defined within expressions.

Identifiers defined within a function can be referenced only within that function; the top level identifiers (global symbols) are visible in all the regions defined on that device. It is important to note that since the symbols can be referenced remotely, top level identifiers can be accessed from other devices using the device set specifications mechanism.

Figure 3-1 shows how the program regions are established and referenced. The functions F11, F12, F13 establish three different regions on device 1. Similarly, the functions F21, F22 establish two different regions on device 2. Any identifier defined within these functions would not be available outside their scope, whereas the symbols defined in either of the top regions would be visible in both the devices.

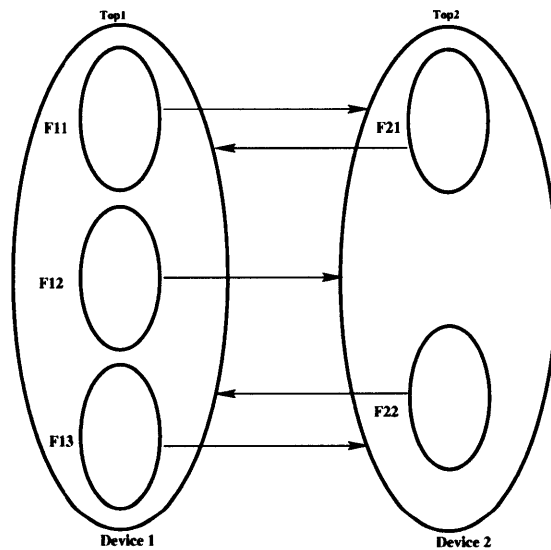


Figure 3-1: Program Regions.

3.3 Compiler Directives

c@t provides two expression types that can be used to modify the behavior of the compiler:

- Since MDES are usually composed of heterogenous devices that are based on different processors, devices and their characteristics must be supplied to the

compiler using device declaration expressions.

- Since functions can be defined on a set of devices, the set can be selected from the declared devices using the Device Set Specification(DSS). DSS is also used to specify target devices of variable and function references.

These compiler directives are discussed in detail here.

3.3.1 Device Declarations

Device declaration expression can be used to define devices that are part of the system.

The structure of device declaration is presented here:

```
(declare-device device ((attr1 val1) (attr2 val2) ... (attrN valN)))
```

```
(declare-cluster (device1 count) (device2 count) ... (deviceN count))
```

Any number of attributes can be specified. The values must be valid literal constants.

The number of devices of each type in the system is specified using declare-cluster expressions.

```
(declare-cluster (device1 count) (device2 count) ... (deviceN count))
```

Device declarations are used for the following purposes:

1. To inform the compiler of the target devices and generate processor-specific machine code.
2. The c@t compiler can use the information to generate code that is most suitable for the target processor configuration and resources.
3. Functions and variables can be selectively defined and referenced based on these attributes.

3.3.2 Device Set Specifications (DSS)

The Device Set Specifications(DSS) are used to select a subset of devices from the set of declared devices. When a variable or a function is defined or referenced, a set

of devices that are targets of definition or of reference can be specified through the device set specification expression.

The device set specification has the form:

```
(@ (grammar g) (count min max)
  (filter expression) (results result-count result-set))
```

In fact, device set specification expression is a separate language that is embedded¹ in c@t. It has four components: grammar, count, filter and results. They are discussed in detail in the next few sections.

Grammar

In MDES, a subset of devices can be selected based on different classifications: resources, roles, locations, structural relations etc.

Fortunately, there are several formalisms for classifying sets of objects. Some formalisms can express some classifications better than others. For instance, complex 3-D structures can be expressed succinctly using Lindenmayer systems; relational algebra can be used for attribute based classifications.

The Grammar field gives the flexibility to use various formalisms for selecting subsets of devices. In the current implementation of c@t, however, only relational expressions can be used, but future extensions can employ other formalisms.

Count

Count specifies the minimum and maximum number of devices to be selected. Both the min and max fields must be positive numbers or metacharacters. The metacharacters can be used when the exact numbers don't matter or can't be known. The valid metacharacters and their special meanings are shown in Table 3.1.

Filter

The Filter field is used to specify the criteria for selecting a subset of devices. The specified filter expression should be valid in the grammar employed. For instance, if

¹It is similar to how regular expressions are embedded in languages such as Perl and Awk.

Character	Meaning
+	1 or more
?	0 or 1
*	0 or more

Table 3.1: Metacharacters of Count Expression.

the grammar used is relational, then a valid combination of logic operators (*and*, *or*, *not*), relational operators ($=$, \neq , $<$, \leq , $>$, \geq), attributes and their values can be used to choose a subset of devices.

The attributes can be any of the properties in the device declaration or any of the variables in scope.

Results

The results field is ignored when used with definitions. It is useful when invoking a remote function or referencing a remote variable. If multiple devices satisfy the DSS and return results, the first of these is returned as the value of that expression and stored as the first element of the result-set array. The remaining results are stored in the subsequent positions of the result-set. The number of results in the result-set is stored in result-count. The application programmer is responsible for defining this array.

3.4 Expressions

Expression types are categorized as primitive or derived. Primitive expression types include variables and procedure calls. Derived expression types are not semantically primitive and can be constructed using the primitive expressions.

3.4.1 Primitive Expression Types

Definitions

A definition should have one of the following forms:

- `(define (device set specification) variable expression)`

- (define (device set specification) variable formals body)

Device set specifications are used to select a subset of devices where variables or functions must be defined.

Variables can be defined either at the top level of a program or at the top level of a function. Functions must be defined only at the top level of a program and nested functions are not permitted.

In the function definition, formals should be a sequence of zero or more pairs of the form (type variable).

A simple function definition is shown here. A function named **square** that returns int and takes an int argument is defined on all **motorola 68000** processors.

```
(define (@ (= processor ``motorola 68000'') ...) int (square int num)
  (* num num))
```

Every program must define a driver function named **main** taking no arguments and returning an integer. Execution of the program would start with this function.

Variable References

An expression consisting of a variable is a variable reference. The value of the variable reference is the value stored in the location to which the variable is bound. It is an error to access an unbound variable.

```
(define (@ ...) int RAMANUJAN_NUMBER 1729)
```

Array variables are defined similarly, except the size of the array is also included in the definition. Also, the initial values are specified as a list. If the initialization list is null, the array elements are assigned appropriate initial values. If the array is of numeric type, its elements are initialized with zeroes; if the array is of any other type, the elements are assigned null characters.

```
(define (@ ...) int[7] primary_primes '(1 2 3 5 7) )
```

Since the array is defined to be of size 7 and only 5 initial values are specified, the last two elements of the array will be initialized to 0s.

The elements of arrays can be referenced using **array-ref** expressions. The structure of **array-ref** expressions is as follows:


```
(array-ref array index-expression)
```

The array elements are numbered sequentially from 0 to the number (array-size - 1). For example, the following expression would return 7.

```
(array-ref primary_primes 4)
```

As previously discussed, both the variables that are defined on the same device and on the other devices can be referenced. However, while referencing the remote variables, device set specifications must be specified to identify the set of devices from which the variable must be referenced.

Literal Expressions

The single quote character is used to include literal constants in c@t code. Numerical constants, string constants, character constants and boolean constants are self-evaluating expressions; they need not be quoted. A few examples are given here:

```
'a  
``Media Lab``  
124  
#t
```

Procedure Calls

The procedure calls have one of the following forms:

- (operator operands) - For calling functions on the same device.
- (operator (device set) operands) - For calling functions defined on other devices.

The device set specification determines the target devices.

The following example shows the difference between calling the function `add` defined on the same device and invoking the one defined on devices that are number-crunchers.

```
(add 3 4)  
(add (@ (filter (= device number-cruncher)) ...) 3 4)
```

Conditionals

An if expression can have one of the following forms:

```
(if test consequent alternate)
```

```
(if test consequent)
```

An if expression is evaluated as follows: first, test is evaluated. If it yields a true value, then consequent is evaluated and its value is returned. Otherwise alternate is evaluated and its value is returned. If test yields a false value and no alternate is specified, then the result of the expression is unspecified.

In the example below, if x is greater than y , x would be returned; else, y would be returned.

```
(if (> x y) x y )
```

In this example, no alternate clause has been specified. If z is 0, 0 would be returned.

```
(if (= z 0) 0 )
```

Assignments

```
(set! variable expression)
```

Expression is evaluated, and the resulting value is stored in the location to which variable is bound. The result of the `set!` is the expression. For example, in the code below, the variable `pi` will be assigned the value of 3.142 and 3.142 will be returned by the expression.

```
(set! pi 3.142)
```

Array Assignments

```
(set-array! array index-expression value-expression)
```

The index-expression and value-expression are evaluated. The value is stored in the array at the location pointed to by the index-expression. For example, the following code would set the fourth element of the array to 11 and return the value 11.

```
(set-array! primary_primes 4 11)
```

3.4.2 Derived Expression Types

Conditionals

```
(cond (clause1) clause2 ...)
```

Each clause should be of the form:

```
((test) (expression))
```

The last clause may be an else clause which has the form:

```
(else (expression))
```

A conditional expression is evaluated by evaluating the test expressions of successive clauses in order until one of them evaluates to a true value. When a test evaluates to a true value, then the remaining expressions in its clause are evaluated in order, and the result of the last expression in the clause is returned as the result of the entire `cond` expression. If all tests evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its expressions are evaluated, and its value is returned.

```
(cond ((> temperature 70) (set! PORTB #x01))
      ((< temperature 40) (set! PORTB #x02))
      (else (set! PORTB #x00)))
```

In this example, if `temperature` is more than 70, the value 1 will be assigned to `PORTB`; if `temperature` is less than 40, the value 2 will be assigned to `PORTB`; if both the above-mentioned conditions are false, `PORTB` will be initialized to 0.

Binding Constructs

The binding construct `let` gives `c@t` a block structure and is used to define local variables within functions.

```
(let bindings body)
```

Bindings should have the form:

```
(type variable init)
```

where each `init` is an expression and `body` should be a sequence of one or more expressions. It is an error for a variable to appear more than once in the list of variables.

The `inits` are evaluated in the current environment in the sequential order, the `body` is evaluated in the extended environment and the value of the last expression of a `body` is returned.

```
(define (@ ...) int (is-circle (int x) (int y) (int r))
  (let
    ((int x_square (* x x))
     (int y_square (* x x))
     (int r_square (* r r)))
    (= (+ x_square y_square) r_square)))
```

This example defines a function called `is-circle` that determines whether its arguments form a circle. The `let` expression is used to define three local variables: `x_square`, `y_square` and `r_square` that are used to check the equation.

Sequencing

```
(begin (expression1) (expression2) )
```

The expressions are evaluated sequentially from left to right, and the values of the last expression is returned. In the example below, `x` will be assigned 6 and the value 7 will be returned.

```
(begin (set! x 6)
      (+ x 1))
```

Iteration

```
(do ((variable1 init1 step1)
    ...
    (variablen initn stepn))
    (test expression)
    command)
```

Do is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a termination condition is met, the loop exits after evaluating the expressions.

Do expressions are evaluated as follows: The init expressions are evaluated in the specified order, the variables are assigned the results of init expressions and then the iteration phase begins.

Each iteration begins by evaluating test; if the result is false, then the command expressions are evaluated in order for effect, the step expressions are evaluated in the specified order, the results of the step expressions are assigned to corresponding variables, and the next iteration begins.

If test evaluates to true, then the expressions are evaluated from left to right, and the value of the last expression is returned.

In the example below, `PORTA` is assigned the values 0 to 254. When `i` becomes 255, the loop is terminated and the `PORTA` is assigned the `STOP_BYTE`.

```
(do ((i 0 (+ i 1)))
    ((= i 255) (set! PORTA STOP_BYTE))
    (set! PORTA i))
```

3.5 Support for Low-level Programming

When programming the embedded systems, it might sometimes be necessary to access specific hardware features. `c@t` provides the following expressions to facilitate such low-level programming.

3.5.1 Special Function Registers and Ports

Application programs can reference the I/O ports and the special function registers as variables. These variables are defined in processor-specific files and they can be included in the program using the `require` expression. For example, the following code sets the `TRISA` register to the value `1F`.

```
(set! TRISA #0x1F)
```

3.5.2 Including Assembly Language Code

Processor-specific assembly language code can be included in the c@t programs using asm expressions.

```
(asm ``movlw 0x25``)
```

It is the responsibility of the programmer to ensure that assembly language code doesn't interact incorrectly with compiler-generated code.

3.5.3 Interrupt Handling

In c@t, interrupts can be handled without writing any assembler code. If a function named `handler` with void return type and no arguments is defined, it will be called directly from the hardware interrupt. In the current implementation, multiple hardware interrupts cannot be handled.

The C compiler that is used with c@t will process this function differently and generates code to save and restore any registers used and exit using the return from interrupt instruction rather than the usual return instructions

Currently, this feature has been tested only for the Microchip PIC midrange processors.

Chapter 4

Runtime Environment

THE c@t runtime environment is designed to ease the process of developing massively distributed embedded applications. Figure 4-1 shows the structure of the code produced by the c@t compiler.

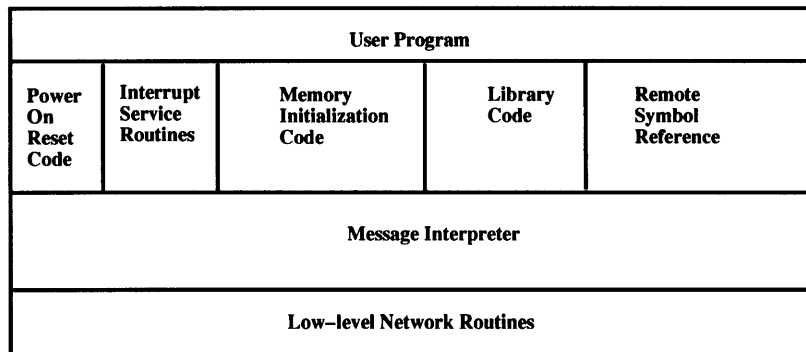


Figure 4-1: The Code Components Generated by the c@t Compiler.

The power on reset code, memory initialization routines, and interrupt service routines provide the basic runtime services. Functions and variables defined on other devices can be referenced using the Remote Symbol Reference (RSR) mechanism. The Associative Naming Scheme (ANS) is supported by the Message Interpreter (MI).

Low-level network routines are responsible for transmission and reception of network messages. They provide the services that are usually offered by the datalink and the physical layers of the seven-layer ISO reference model [17]. They may also need to perform some of the tasks of higher layers such as packet fragmentation, packet assembly, packet sequencing, end-to-end message deliveries, etc. Currently,

these low-level routines are not implemented in c@t. However, we believe that it should be possible to add these low-level routines to the c@t runtime environment.

Conventional distributed systems are built using a standard layered model. However, as D. Clark et al [10] remark, although the layered model facilitates modular development of subsystems, they also impose inessential constraints and lead to inefficient implementations. Since MDES have minimal resources, they need to be programmed using efficient strategies. As Figure 4-1 shows, the code produced by the c@t compiler is vertically integrated. That is, the c@t compiler not only produces the application code, but every single code component that runs on that device. This integrated approach can lead to efficient code realizations, since all the components can be tailored to the needs and characteristics of the application. Further, since the application and the supporting network routines are generated together, the compiler can choose the best possible solution architecture for the given application as explained in Section 1.2.3.

This chapter explains design and implementation of runtime in detail.

4.1 Runtime Design

As explained in Section 1.1, developing MDES applications can be difficult due to the inherent interaction complexity. c@t provides the RSR mechanism and the ANS to alleviate the interaction complexity. c@t supports the ANS through the Message Interpreter (MI). The following sections present the design of RSR and MI.

4.1.1 Remote Symbol Reference Subsystem

The idea of RSR is based on the observation that function calls and variable references are well-known and well-understood mechanisms for transfer of control and data within a program running on a single device. Therefore, it is proposed that this same mechanism be extended to provide for transfer of control and data across a communication network. There are several alternatives to RSR: message passing model (employed by SR [5], for example) and the tuple space model (employed by Linda [9]). I believe that a choice between these alternatives is not significant, as the

problems of reliable and efficient communications are quite similar to the problems encountered by the Remote Symbol Reference paradigm used by this work. The overriding consideration that made me choose the RSR is that many popular languages use the function calls and variable references as the mechanisms for data and control transfer mechanisms. So, it could be easy for programmers to learn and use the language, if necessary.

The RSR is inspired by the Remote Procedure Calls (RPC) mechanism [7]. But, there are some significant differences between RSR and RPC:

1. RPC implementations require several manual operations such as installing stubs, registry etc, whereas in c@t, the application programmer is not required to perform any such additional tasks. This is not to say that it is an unique feature of c@t. In fact, even the early distributed programming languages such as NIL had [29] provided such a convenient interface for remote method invocations; but, these conventional distributed programming languages don't have all the features necessary to support MDES programming.
2. In RPC and other conventional distributed programming implementations, it is usually assumed a remote function invocation would be executed on a single target device. In c@t, a function could be executed on multiple target devices with a single request. This is required for programming MDES, due to the equivalence sets of devices that form the system.
3. While using RPC, syntactically, there is no difference between calling a remote procedure and a local procedure. In c@t, there is a minor difference - using DSS the target devices need to be specified while referencing remote symbols. The following code snippet illustrates this difference:

```
; calling local function
(square 5)

; calling remote function
(square (@ (= device calculator) ...) 5)
```

RSR Sequence

When a remote function is invoked or a remote variable is referenced, the calling environment is suspended, the access request and parameters, if any, are passed across the network to the environments (which are referred to as callees) where the function is to be executed or the variable is to be accessed and the desired action is taken on those devices. When the callees finish and produce their results, those results are passed back to the calling environment, where execution resumes as if returning from a single-machine transfer. Figure 4-2 presents these steps.

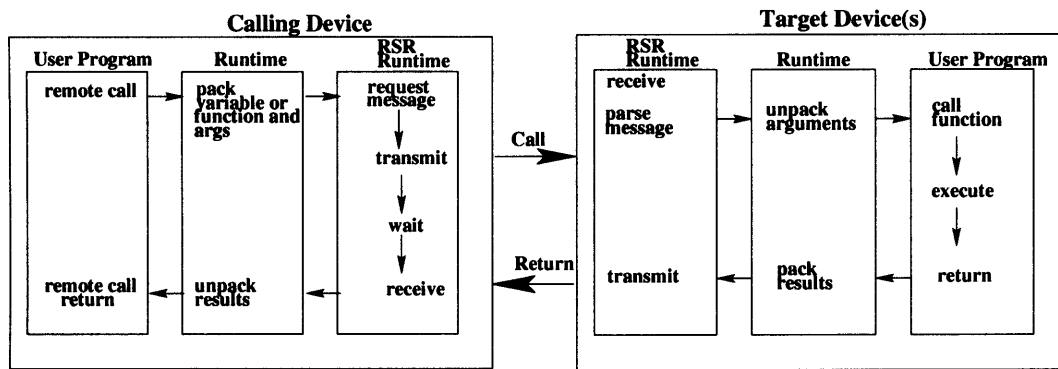


Figure 4-2: The RSR Components and Their Interactions for a Simple Call.

The actual sequence of events in invoking a remote function is more elaborate when the required number of target devices is more than one. The number of results expected is specified in the count part of the DSS. The calling device waits until the specified number of target devices return results. For instance, while executing the following code, the caller would wait for at least 5 (since the minimum specified by the count component is 5) devices to return results before transferring control back to the function that initiated the remote call.

```
(set-seed (@ (count 5 15) (filter (= device random-gen)) ...) 5)
```

Figure 4-3 shows the sequence of events in referencing a remote symbol on multiple target devices. The process starts with the calling device sending out the “request for response” message. The devices that match the filter return a “ready to respond” message back to the calling device. The calling device selects (based on some cost function) a subset of devices from the ones that responded and sends the “execute”

message. The selected devices execute the requested action and return results. The first result that is received is returned as the value of the function call and the remaining results are stored in the specified result-set.

If the remote symbol referenced is a variable, the target devices just return the variable value and skip the “ready to respond” and “execute” message phases.

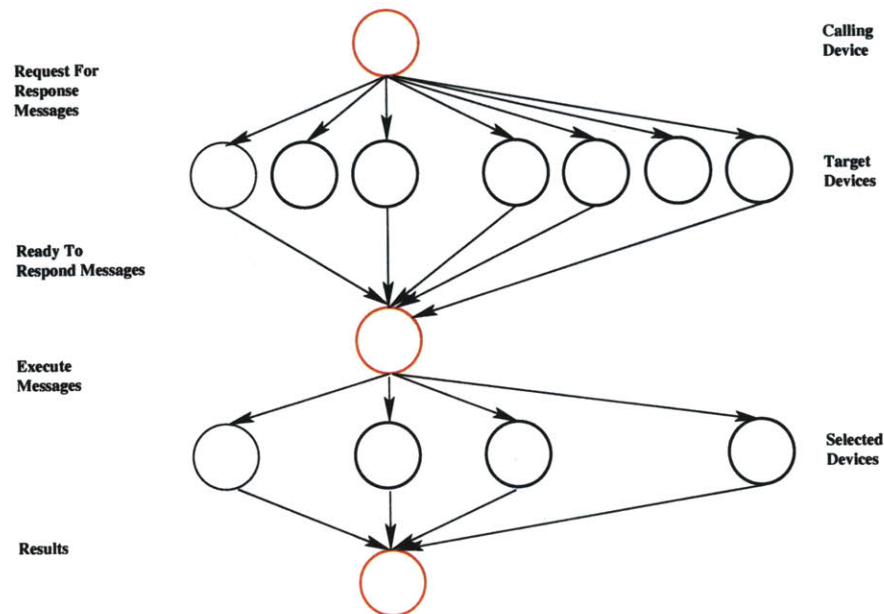


Figure 4-3: The Sequence of Events in Invoking a Remote Function.

There are a few caveats due to the asynchronous nature of execution and the potential device failures. First, there is the problem of synchronization: a target device might satisfy the filtering criteria when receiving the “request for response” message and before it receives the “execute” message, its state could change such that it fails the DSS filter. When the calling device sends the execute message to the device with the changed state, that device would send a “refuse to execute” message to notify the calling device about its state change. Then the calling device would repeat the entire execution sequence.

Second, there is the problem of device failures; the devices could just stop working at any time. If a device stops, before it responds to “request for response” message, there is nothing to be done to handle the failure. However, if it fails after sending the “ready to respond” message, the calling device would repeat the entire execution

sequence.

It is ensured that while repeating the sequence, the devices that return results in the first try would be eliminated to avoid redundant actions on those devices.

This execution sequence and fault tolerance model can handle only the stopping failures and not the Byzantine failures.

4.1.2 Message Interpreter

The Message Interpreter performs the following two functions:

1. Parsing device set specifications - When a remote function is invoked or a remote variable is accessed, a message with target devices expressed as the Device Set Specifications (DSS) is transmitted. Every device that receives the packet parses the DSS expression to determine whether the message is intended for it.
2. Parsing symbols - The calling device also specifies the function to be executed or the variable value to be returned. MI is responsible for parsing the arguments, if any, and the return type to ensure that specified types and the local symbol types match¹. For example, in the code below, the integer variable `current-temperature` on heater is assigned the value returned by the function `get-current-temperature` defined on sensors. The sender would have specified that an integer return value is expected. But, the return type of the function `get-current-temperature` is string. So, the parser on sensor would catch this mismatch and prevent the sensor from responding to the request initiated by heaters.

```
(define (@ (= device heater) ...) int current-temperature 0)
(define (@ (= device sensor) ...) string temperature ``60°C``)
(define (@ (= device sensor) ...) string (get-current-temperature)
        temperature)
(define (@ (= device heater) ...) void (check-temperature))
```

¹The c@t compiler includes the symbol table to the generated code for the use of these parsers.

```
(set! current-temperature
  (get-current-temperature (@ (= device sensor) ...)))
```

This example illustrates an important point - the remote symbol references cannot be statically type checked as the same symbol can be defined to be of different datatypes on different devices and the exact target devices on which the symbol would be referenced are not known at the compile time.

It is important to see that the message interpreter might contain multiple (one for each grammar used) parsers for evaluating device set specifications.

4.2 Runtime Implementation

The runtime environment has to be developed individually for every target processor. Currently, the runtime is implemented only for Microchip [24] mid-range processors. In particular, this implementation has been tested on the microcontroller - Microchip PIC 16F84. This microcontroller has 1K words of code memory, 68 bytes of RAM, 64 bytes of ROM and a 8-level stack that can store only the program counter.

The primary reason for testing the implementation on PIC 16F84 is that if this language can be used with such a resource-constrained device, it is probable that it could be used for programming processors with more memory and processing power.

The next sections present the implementation details of the runtime environment.

4.2.1 Remote Symbol Reference Subsystem

Implementing the RSR described in Section 4-2 requires RAM for buffering the intermediate results and the code memory for processing code. Since PIC microcontrollers have limited RAM and ROM, the RSR subsystem must be simplified to work on these devices. The simplifications are:

- While referencing remote-symbols, the count expressions are ignored. That is, all the devices that satisfy the specification would execute the requested action and return results.

- If multiple devices return results, a maximum of five results are preserved.

4.2.2 Message Interpreter

The limited computational resources impose some limits on the message interpreter; it can parse only simple relational expressions and the messages can be a maximum of 18 bytes long.

The message structure is given in Table 4.2. Consider a specific example: if the following line of code is defined on a lamp whose ID is 25 and its current running packet sequence number is 61, the message would be as shown in Table 4.1.

```
(add_lamp (@ ... (filter (and (= device ``switch'') (< hop 5)))) id)
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	5	'c'	'a'	'='	'd'	0	1	'<'	'h'	0	5	'a'	'2'	'5'	0	6	1

Table 4.1: A Message Example.

The limited computational resources of the microcontroller and the small message size (18 bytes) impose the following limitations on the applications. It is important to understand that these are not the limitations of the language or the compiler, but the runtime environment that is implemented for PIC 16F84.

1. The device set specifications can only be relational expressions. Further, these expressions can contain at the most two sub-expressions. The following filters are valid:

```
(and simple-exp1 simple-exp2)
```

```
(or simple-exp1 simple-exp2)
```

```
(not simple-exp)
```

```
(simple-exp)
```

The simple expressions can use only the following four operators: '<', '>', '=', and '!' (represents not equal).

2. Only device types and integer variables can be used as attributes in simple expressions.

3. Only remote functions that return no values or integers and take no arguments or one integer argument can be invoked. Similarly, only remote variables that are of type integers can be accessed.
4. The most constraining restriction is that two different symbols (variables or functions), which are defined on the same device, cannot have names with the same initial letter. This is because when these symbols are transmitted by the RSR, only their first letter is used to represent them (please see Table 4.2). For example, if a device has a variable named **temperature**, it would be represented using the initial letter 't'.

Despite these limitations, useful distributed embedded applications can be developed, as illustrated by the sample application presented in chapter 5.

Bytes	Field	Details
0 - 1	Source ID	The ID of the message source.
2	Message Type	This byte can contain only two values - 'c' and 'r'. The character 'c' indicates that this is a request message and 'r' indicates this is a reply message.
3 - 11	Device Set Specification	The relational expression that specifies the target devices. The individual bytes and their details are: 3 The values can be 'a', 'o', 'n' and '='. The first three represent the logical operations conjunction, disjunction and negation. The character '=' represents a simple expression. 4 The values can be one of the characters: '=', '!', '<', and '>'. They represent the relational operators equal, not equal, less than and greater than. 5 A single character that represents attributes. 6 - 7 Contains the attribute value. 8 Same as the fourth byte. But, this byte can be null, if this is a simple expression. 9 Same as the fifth byte. But, this byte can be null, if this is a simple expression. 10 - 11 Same as the sixth and seventh bytes. But, this byte can be null, if this is a simple expression.
12	Remote Symbol or Return Values Flag	In a request message (byte2 = 'c'), this byte contains the variable or function name. In a reply message (byte2 = 'r'), this byte can contain the values 0 or 1. The value 0 means no return value in the message and 1 means a value is returned.
13-14	Arguments or Return Values	In a request message, these bytes contain the arguments to the remote function. In a reply message, these bytes contain the return values, if any, or null.
15	Hop Count	Number of hops completed by this message. Devices increment this field every time they forward the message.
16-17	Message ID	This field is used by devices to distinguish between different messages they have transmitted.

Table 4.2: The Message Structure in the Runtime Environment for Microchip PIC 16F84.

Chapter 5

Application Example

TO evaluate the c@t language, a simple self-organizing building control application called Get-Set-Go has been developed in c@t. A simulation environment has also been implemented to run and evaluate such distributed embedded systems.

This chapter presents the details of both the application and the simulation environment. It also discusses the lessons learned from developing this application.

5.1 Simulation Environment

The distributed embedded systems simulator is developed using a two-layered architecture, as shown in Figure 5-1. The two layers are:

1. Microcontroller Layer - The microcontroller layer simulates the functionality of a Microchip PIC 16F84 microcontroller. It implements the functions and features provided by the 16F84. This layer takes a Intel hex file as input and executes the program in that file.
2. Interface Layer - The interface layer performs two functions:
 - (a) Acts as a conduit between users and the microcontroller layer. It notifies the microcontroller of all user actions and presents the microcontroller output to the user. That is, when a user presses the switch button, this layer notifies the underlying microcontroller by pulling down the input pin

13. When the voltage level of pin 3 changes, it updates the display to reflect the status (if pin is high, lamp is on and if pin is low, lamp is off).
- (b) Provides the low-level message handling services for the devices. This function is explained in detail below.

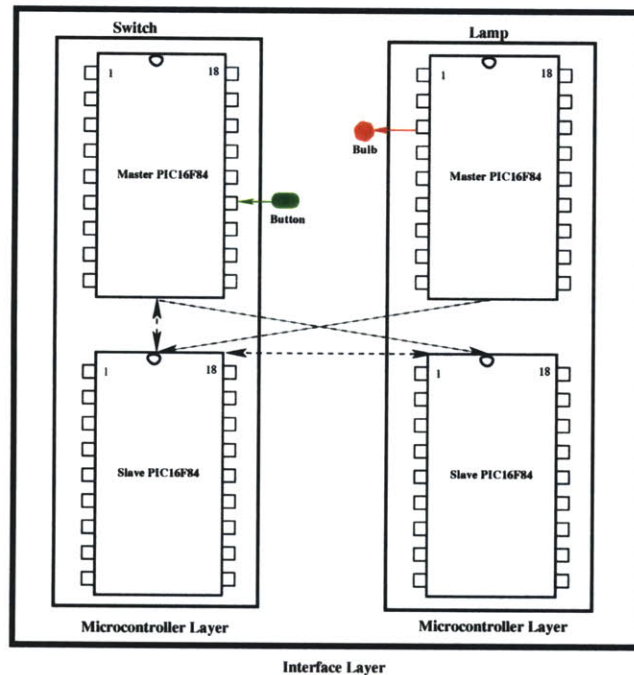


Figure 5-1: Simulator Architecture.

In the initial versions of the simulator, a microcontroller layer unit was realized as a single simulated PIC 16F84. The program memory (1k words) and the RAM (68 bytes) available on a PIC 16F84 turned out to be inadequate for the Get-Set-Go system. So, as shown in Figure 5-1, the microcontroller layer unit was modified¹ to be realized using two PIC 16F84's - a master and a slave. The master runs the application tasks and the slave handles the network messages.

It is important to note that the simulator contains a single interface layer that is comprised of several instances (one per simulated appliance) of the microcontroller layer. The Interface layer simulates the low-level network mechanisms: it transfers

¹The c@t compiler and the runtime were also modified for this configuration.

the messages between slave and master, slave and neighboring units, and master and neighboring units.

When the master and the slave want to communicate with each other, they write the message to a designated area in RAM and flip a prespecified location in RAM to inform the interface layer that a message needs to be transmitted. Similarly, if a master or a slave wants to communicate with other units, they write the message to a designated area in RAM and flip another prespecified location in RAM to notify that a message needs to be transmitted to the neighboring (any device at a one hop distance) devices. The Interface layer reads the message off the designated memory area and sends it to that unit's neighbors. When the interface layer needs to pass the message to the microcontroller layer unit, it writes the message in the designated area and generates an interrupt to let the corresponding microcontroller know that the data is available.

As illustrated in Figure 5-1, both the slave and the master can send messages, however only the slave can receive messages from other units. The slave parses the incoming messages to determine whether that unit satisfies the DSS filter specified in the messages. If it does, it transfers the message to the master to execute the actions specified.

A screen shot of the simulator is presented in Figure 5-2. The simulator takes the number of devices as input and creates a randomly distributed network of devices.

5.2 Get-Set-Go: A Self-organizing Building Control System

Get-Set-Go is a simple building control application consisting of 50 switches and 100 lamps. In this application, devices organize themselves into a network such that the following two conditions are met: First, a switch controls exactly one lamp in its immediate neighborhood. Second, a lamp may be associated with more than one switch in its immediate neighborhood.

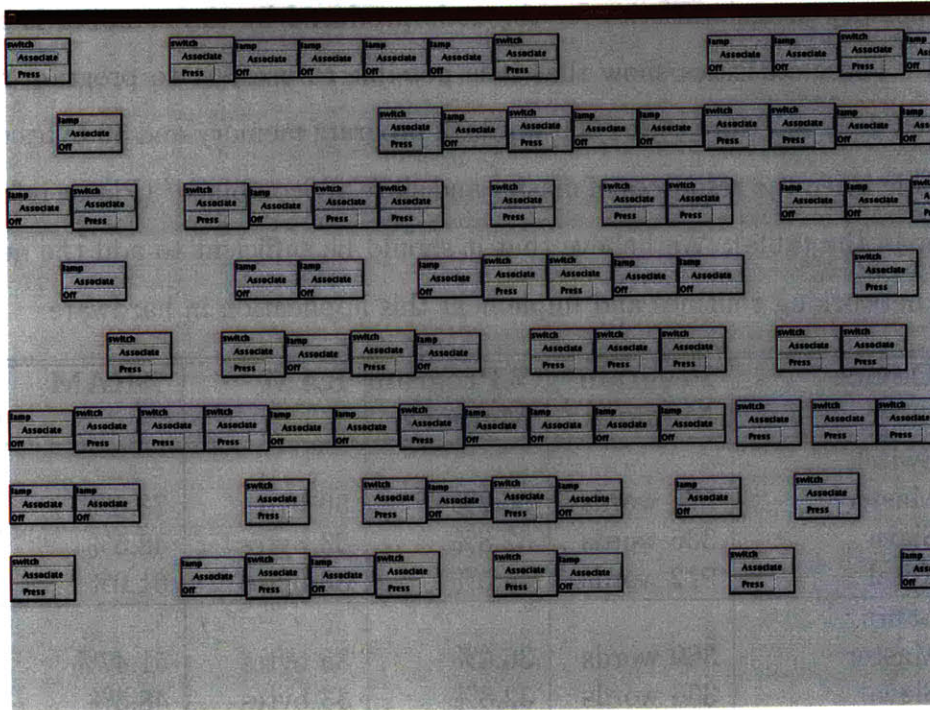


Figure 5-2: Simulator Screen Shot.

This application works in two phases:

1. Initialization phase - Lamps register themselves with the neighboring switches. If there are no switches in the neighborhood, that lamp is left without any association. If more than one lamp registers with a switch, the association with the lamp that registers last will be preserved.
2. Execution phase - Switches respond to user actions; when a switch is pressed, it toggles the lamp that it controls. If it is not associated with any lamp, it ignores the user actions.

This application has been implemented in c@t and the source code is given in Appendix B.

The c@t compiler produces executables that run in the simulator described above. For each switch and lamp, the compiler produces two executables - one for the master unit and one for the slave unit. For the given source code, it produces 300 (50 switches and 100 lamps) executable files.

The memory usage statistics for the code produced by the compiler is given in Table 5.1. These statistics show that it is possible to use c@t to program resource constrained devices. More than 1000 words of program memory and 50 bytes of RAM are available when the resources of master and slave are combined² (Please refer to the totals row in the table). We believe that it should be sufficient to add the necessary low-level networking routines and implement this application in hardware.

Device	Program Memory	%Program Memory	RAM	%RAM
Switch				
Master	576 words	56.2%	50 bytes	73.5%
Slave	336 words	32.8%	33 bytes	48.5%
Total	912 words	44.5%	83 bytes	61.0%
Lamp				
Master	369 words	36.0%	35 bytes	51.47%
Slave	336 words	32.8%	33 bytes	48.5%
Total	705 words	34.4%	68 bytes	50.0%

Table 5.1: Memory Usage Statistics.

This Get-Set-Go system works in the simulation environment. The devices are able to set associations and respond to user events. When the switch button is pressed, the associated lamp is turned off, if it was on and it is turned on, if it was off.

Although this application is simple, the devices perform a few interesting functions:

- Service discovery - Lamps locate the switches in a completely decentralized environment.
- Message routing and parsing - Devices parse messages to determine whether they are intended to be recipients of those messages and perform the specified actions.
- Event notification - Switches notify the lamps of the events and change their states.

²Microchip PIC 16F628 has 2048 words of program memory, 136 bytes of RAM and 128 bytes of RAM. These numbers are exactly equal to the sum of the resources of two PIC 16F84's.

This application demonstrates that the c@t language can be used to build distributed embedded systems. Further, these applications can be executed on devices with minimal resources.

Chapter 6

Conclusions and Future Work

THIS thesis presented c@t, a language for programming distributed embedded systems. The language uses the principles of collective programming, declarative style network programming, associative naming and device set specifications to address the scale and interaction complexities of massively distributed embedded systems.

The c@t compiler is written in Scheme and amounts to 3000 lines of code. The language runtime environment has been implemented on Microchip PIC midrange microcontrollers. In particular, this language can be used to write applications for tiny processors like PIC 16F628 which has only 2K of program words, 136 bytes of RAM, 128 bytes of ROM, and 8-level stack that can store only the program counter during control transfers. The runtime environment is written in C and PIC assembly language and has about 250 lines of code.

A software system that can simulate PIC 16F84 based networked embedded systems has also been completed. This system contains about 3000 lines of C++ code and 700 lines of Java code.

The completed work focuses on the basic features of the c@t language, the compiler, and the runtime environment. We are still in the early stages of acquiring experience with the use of c@t. Although c@t was used to write several simple applications, it has not been used in a full-scale project.

Based on the current experience with c@t, it is clear that the language needs a few basic improvements: expressions for specifying the network topology and characteristics, making device set specifications as first class objects in the language, support for dynamic arrays, better support for string operations, case statements, lambda abstractions, low-level network libraries, etc.

The most significant feature missing is the offline analysis for producing vertically integrated systems. Consider an embedded device network presented in Figure 6-1. As shown, if the processor resource details (the tuple (p, m, b) represents processor configuration, memory, and battery) and link costs are available, the compiler can analyze and produce the most efficient implementation.

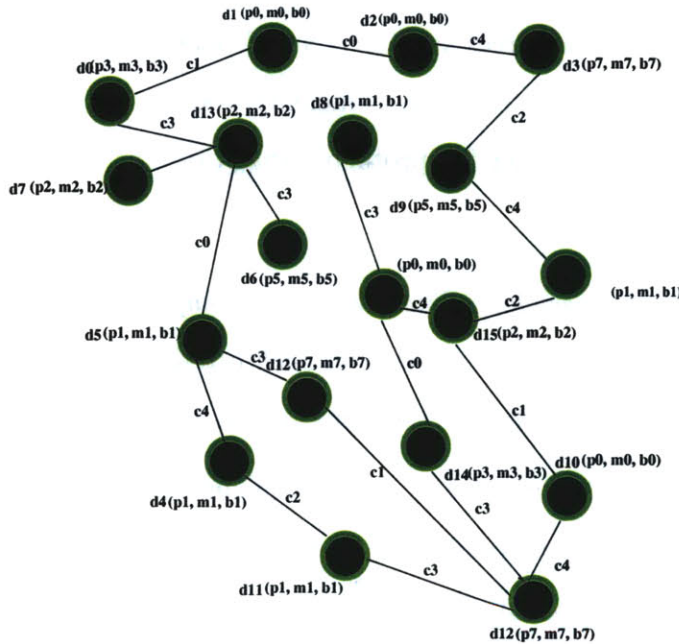


Figure 6-1: An Embedded Device Network.

We believe that vertical integration is possible. Since all the devices are programmed together and their interactions are specified in a declarative style, the compiler can define computations and network architectures that are most suitable for the application. For instance, if the device d7 needs to use a function named f_1 , the compiler can define that function on d7 or on any of the other devices. The choice between local definition and remote definition can be made based on the number of devices that need to invoke that function, memory available on the devices, message

costs, etc. Fortunately, several network flow analysis algorithms and techniques [4] that can be applied here are available.

One might object to this vertical integration approach for any of the following reasons:

- *Is it necessary to perform such extensive offline analysis to produce efficient implementations? As the processor costs are going down, is this approach beneficial?*

We believe this approach can be useful. When the compiler tries to optimize, it is trying to reduce space complexity, time complexity and message complexity. Due to continuous reduction in memory costs, space complexity may not be of concern. However, resources such as channel bandwidth and battery power are valuable and they need to be used efficiently. As demonstrated by Heinzelman et al [15], there is a need for protocols that are efficient; in terms of energy, power consumption, etc.

- *Is this approach possible? Since the scale complexity is an inherent part of MDES, would it be possible to perform such extensive analysis? Would there be sufficient resources on the workstations to run such complex algorithms?*

We don't know. Since neither we nor anyone else has implemented such a system, it is difficult to predict. However, as per Moore's law - the processor costs are going down with simultaneous increase in processing power. Given such abundant workstation resources, we can use them to generate efficient code for resource constrained devices.

Certainly, the language can be enhanced in multiple ways to make it a more effective tool. However, as the successful implementation of the sample application Get-Set-Go demonstrates, even the current version can be used to develop meaningful applications. Although this application is simple, it performs several interesting tasks: service discovery, network self-configuration, event notification, message parsing and message routing.

More importantly, since it has been shown that this language can be used to develop applications for a tiny microcontroller, it will also be useful for developing applications on processors with better computational resources.

Appendix A

Formal Syntax

This section presents a formal syntax of for c@t written in an extended BNF. The following extensions to BNF are used to make the description more concise: [*item*] means zero or one occurrence of < *item* >, < *item* >* means zero or more occurrences of < *item* >, and < *item* >+ means at least < *item* >.

Lexical Structure

```
<token> → <identifier> | <boolean> | <number> |  
         <character> | <strings> | ( | ) | '  
<comment> → ; (all subsequent characters up to line break)  
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
<letter> → a | b | c | ... | z  
<whitespace> → space | newline  
<atmosphere> → <whitespace> | <comment>  
<delimiter> → <atmosphere> | ( | ) | ``  
<intertoken space> → <atmosphere>*  
<identifier> → <initial><subsequent>*  
<boolean> → #t | #f <initial> → <letter>  
<subsequent> → <initial> | <digit> | <special subsequent>  
<special subsequent> → . (underscore)  
<syntactic keyword> → <expression keyword>
```

```

    | else | define | declare-device | declare-cluster
<expression keyword> → quote | if | set! | begin
    | cond | and | or | let | do
<variable> → (any <identifier> that isn't also
    a <syntactic keyword>) | (any <identifier> that isn't also
    a <syntactic keyword>) <dss>
<dss> → (@ (grammar <grammar-type>) (count <min> <max>))
    <filter> (results <result-count> <result-set>))
<grammar-type> → relational
<min> → <digit>+
<max> → <digit>+
<filter> → <simple filter> | <compound filter>
<simple filter> → (<rel-op> <identifier> <simple datum >)
<rel-op> → == | != | < | <= | > | >=
<compound filter> → <conjunction> | <disjunction > | <negation>
<conjunction> → (and <filter> <filter>+)
<disjunction> → (or <filter> <filter>+)
<negation> → (not <filter>)
<result-count> → <identifier>+
<result-set> → <identifier>+
<character> → #\ (any character) | #\ <character name>
<character name> → space | newline
<string> → `` <string element>* ``
<string element> → (any character other than `` or \ )
    | \" | \\
<number> → <num 10> | <num 16>

```

The rules for <num R>, <real R>, <ureal R>, <uninteger R>, and <prefix R> should be replicated for R = 10, 16. There are no rules for <decimal 16>, which means that numbers

containing decimal points or exponents must be in decimal radix.

<num R> → <radix R> | <real R>
<real R> → <sign> | <ureal R>
<ureal R> → <uinteger R> | <decimal R>
<decimal 10> → <uinteger 10> | .<digit 10>+
 <digit 10>+ . <digit10>*
<uinteger R> → <digit R>+
<radix 10> → <empty> | #d
<radix 16> → #x
<digit 10> → <digit>
<digit 16> → <digit> | a | b | c | d | e | f

Datum

<datum> → <simple datum> | <compound datum> |
<simple datum> → <boolean> | <number>
 | <character> | <string> | <symbol>
<symbol> → <identifier>
<compound datum> → <array>
<array> → (<datum>*)

Expressions

<expression> → <variable> | <literal> | <procedure call>
 | <conditional> | <assignment> | <derived expression>
<literal> → quotation | self-evaluating
<self-evaluating> → boolean | number | character | string
<quotation> → '<datum> | (quote <datum>)
<procedure call> → (<operator> [*dss*] <operand>*)
<operator> → <expression>
<operand> → <expression>
<conditional> → (if <test> <consequent> <alternate>)
<test> → <expression>

<consequent> → <expression>
 <alternate> → <expression> | empty
 <assignment> → (set! <variable> <expression>)
 <derived expression> →
 (cond <cond clause>+)
 | (cond <cond clause>+ (else <sequence>))
 | (and <test>*)
 | (or <test>*)
 | (not <test>)
 | (let (<binding spec>*) <body>)
 | (begin <sequence>)
 | (do (<iteration spec>*) (<test> <do result>)
 <command>*)
 <cond clause> → (<test> <sequence>)
 <binding spec> → (<variable> <expression>)
 <iteration spec> → (<variable> <init> <step>)
 | (<variable> <init>)
 <init> → <expression>
 <step> → <expression>
 <do result> → <sequence> | empty

Programs and Definitions

<program> →
 <device>+ <cluster>+ <definition>+
 <device> →
 (declare-device <dev-name> ((processor <string>)
 (ram <string>) (code-mem <string>) (rom <string>)
 (<property> <simple datum>)*)
 <dev-name> → <identifier>
 <property> → <identifier>
 <cluster> →

```

        (declare-cluster <clust-name> (<dev-name> <count>)+)
<cluster-name> → <identifier>
<count> → <digit>+
<definition> →
    | (define <dss> <type> <variable> <expression>)
    | (define <dss> <type spec> <variable> <def formals> <body>)
<type spec> → void | <type>[<digit>+]*
<type> → char | int | float | string
<def formals> → <variable>*
<body> → <definition>* <sequence>
<sequence> → <command>* <expression>
<command> → <expression>

```

Appendix B

Source Code

```
; declare devices and cluster
(declare-device switch ((processor "16f84")))
(declare-device lamp ((processor "16f84")))
(declare-cluster gsg ((switch 50) (lamp 100)))

;; ----- Switch Code -----
;define variables on switches.
(define (@ () (= device "switch")) int lamp-count 0)
(define (@ () (= device "switch")) int lamp-list 0)
(define (@ () (= device "switch")) int on 0)
(define (@ () (= device "switch")) int prevrb6 1)

;define functions on switch.
(define (@ () (= device "switch")) void (handler)
  (if (and (= rbie 1) (= rbif 1))
    (if (and (!= rb6 prevrb6) (!= lamp-count 0))
      (begin
        (set! gie 0)
        (set! rbif 0))
    )
  )
)
```

10


```

    (set! prevrb6 rb6)
    (if (= on 0)
        (begin
            (set! on 1)
            (activate (@ (count 1 1) (grammar 'relational)
                        (filter (= id lamp_list))))
            (begin
                (set! on 0)
                (deactivate (@ (count 1 1) (grammar 'relational)
                              (filter (= id lamp_list))))
                (set! gie 1))))
        (set! gie 1))))
30
(define (@ () (= device "switch")) int (add_lamp (int id))
  (begin
    (set! lamp_list id)
    (set! lamp_count (+ 1 lamp_count)))
  0)
(define (@ () (= device "switch")) int (main)
  (begin
    (set! lamp_count 0)
    (set! rbie 1)
    (set! gie 1)
    (do ()
      ()
      ()))
  7))
;; ----- End of Switch Code -----

;; ----- Lamp Code -----
;define functions on lamp.

```

```

(define (@ () (= device "lamp")) int (activate)
  (set! ra4 1)
(define (@ () (= device "lamp")) int (deactivate)
  (set! ra4 0)
(define (@ () (= device "lamp")) int (notify_switch)
  (add_lamp (@ (count 1 1) (grammar 'relational)
    (filter (and (= device 0) (= hop 1)))) i)
(define (@ () (= device "lamp")) void (handler)
  5)
(define (@ () (= device "lamp")) int (main)
  (set! rbie 1)
  (set! gie 1)
  (set! ra4 0)
  (set! trisa4 0)
  (notify_switch)
  (do ()
    ()
    ())
1)
;; ----- End of Lamp Code -----

```

Bibliography

- [1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.
- [2] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.
- [3] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. *Proc. 17th SOSP*, 1999.
- [4] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows*. Prentice Hall, 1993.
- [5] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nielsen, T. Purdin, and G. Townsend. An overview of the sr language and implementation. *ACM Transactions on Programming Language Systems*, 10(1):51–86, 1988.
- [6] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.
- [7] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [8] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.
- [9] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in linda. In *Symposium on Principles of Programming Language*. ACM, New York, 1986.

- [10] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Symposium on communication Architectures and Protocols*, pages 200–208. ACM Press, september 1990.
- [11] D. Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, 1999.
- [12] Echelon corporation. <http://www.echelon.com>.
- [13] P. Eles, K. Kuchcinski, and Z. Peng. *System Synthesis with VHDL*. Kluwer Academic Publishers, 1998.
- [14] N. Gershenfeld. *When Things Start to Think*. Henry Holt & Company, Inc., 1999.
- [15] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocols for wireless microsensor networks. In *Proc. Hawaaiian Int'l Conf. on Systems Science*, 2000.
- [16] Hi-tech picc lite compiler. <http://www.htsoft.com>.
- [17] Information processing systems - open systems interconnection - basic reference model. ISO-7498, 1984.
- [18] T. Lawrence, J. Suominen T. McLeish, and K. Larson. House_n. http://architecture.mit.edu/house_n/.
- [19] G. Leavens. Introduction to the literature on programming language design. <http://www.cs.iastate.edu/~leavens/homepage.html>, 1999.
- [20] C. Leopold. *Parallel and Distributed Computing - A Survey of Models, Paradigms, and Approaches*. John Wiley & Sons, Inc., 2001.
- [21] J. Lifton. Pushpin computing. Unpublished master's thesis available at <http://www.media.mit.edu/~lifton/Pushpin/>, 2001.

- [22] A. Lindenmayer. Mathematical models for cellular interaction in development, parts i and ii. *Journal of Theoretical Biology*, 18:280–315, 1968.
- [23] B.B. Mandelbrot. *The fractal geometry of nature*. W.H. Freeman and Company, 1977.
- [24] Microchip technology inc. <http://www.microchip.com>.
- [25] R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, 2001.
- [26] Plt scheme. <http://www.plt-scheme.org>.
- [27] D. Seetharam, H. Shrikumar, T. Lawrence, T. McLeish, and K. Larson. Distributed building networks. http://architecture.mit.edu/house_n/.
- [28] H. Shrikumar. Data composability in myriad nets (invited talk): De-layering in billion node mobile networks. In *Second ACM international workshop on Data engineering for wireless and mobile access*, pages 43–43, 2001.
- [29] R. E. Strom and S. Yemini. NIL: An integrated language and system for distributed programming. In *SIGPLAN'83*, pages 73 – 82, June 1983.
- [30] H. Takada and K. Sakamura. Compact, low-cost, but real-time distributed computing for computer augmented environments. In *IEEE Computer Society Workshop on Future Trends of Dist. Comp. Sys.*, pages 56–63, Aug. 1995.
- [31] D. Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, May 2000.
- [32] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.
- [33] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.