Towards RSEAM: Resilient Serial Execution on Amorphous Machines

by

Andrew Sutherland

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by . . . .
Jonathan Bachrach
Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Towards RSEAM: Resilient Serial Execution on Amorphous Machines

by

Andrew Sutherland

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

An amorphous computing substrate is comprised of hundreds or thousands of individual computing nodes, each with limited resources. Amorphous computers have primarily been used for sensor fusion purposes, with nodes coordinating to aggregate, log, and act on data gathered. Although several programming models exist for these tasks, they are frequently mismatched with the problem domain, and do not scale. There are no existing languages that provide a way to reliably execute programs that far exceed the capabilities of any single computing node. We introduce a language RGLL that is both well suited to amorphous computing and scales well. We describe a hypothetical language, RSEAM, that can compile large programs not designed with amorphous computers in mind so that they can resiliently execute on amorphous computers. We describe the mechanisms of its operation, and show how they can be implemented in RGLL.

Thesis Supervisor: Jonathan Bachrach
Title: Research Scientist

Acknowledgements

I would like to thank the following people and/or entities:

- The Dynamic Languages Group at the MIT Artificial Intelligence Lab for funding me and providing me the opportunity to try and wrap my head around the functional programming idiom.
- Jonathan Bachrach for being an excellent supervisor, thesis or otherwise, and constant source of enthusiasm.
- Greg Sullivan for being a great office-mate and a man to turn to when it comes to the many mysteries of emacs.
- James Knight for his thesis commentary and help on understanding the intricacies of Goo.
- Mike Salib for his excellent and thorough thesis proof-reading and commentary.

Contents

Chapter 1

Introduction

1.1 Background

Continual progress in both miniaturizing computing components and decreasing the cost of those components is paving the way for a new mode of computing, better suited to tasks that were not previously feasible to computers. The new mode of computing has been given many different names by many different people: paintable computing, amorphous computing, swarm computing. All of these names are talking about the same thing, massive numbers of small, cheap computational devices working together. Communicating via a broadcast medium such as radio or infrared, these otherwise homogeneous computing devices organize themselves to perform the task set before them. What is new about this approach is that it deviates from traditional computing strategy. In nature, there are two main strategies of species survival: quality, and quantity. Human beings go for quality, producing (most of the time) one offspring at a time. They then take great care to rear and protect this offspring. In contrast, other species, such as sea turtles, lay a couple hundred eggs, leave, and hope that more than a few survive. Traditional computing strategy has been one of quality. We produce relatively few computing devices, and if they fail, we are sunk. The new mode of computing, which I will hereafter refer to solely as amorphous computing, is not dependent on the continued survival of any one, five, or ten computing nodes for successful operation.

A key side effect of spreading our resources around rather than concentrating them is that our computing nodes will be far less capable than your average desktop computer. Rather than having a processor measured in gigahertz and memory measured in hundreds of megabytes, we get computing nodes with processor speeds in tens of megahertz and memory in tens of kilobytes. Our communications bandwidth is also limited, and implicitly shared among our neighbors. The current state of radio broadcast on shipping hardware is actually quite poor, but this will most likely see the greatest improvement in the future, so I won't focus on it. As is implied by the analogy to sea turtles and their offspring, our hardware is not going to be as reliable as a desktop PC; however, this requires clarification. In the process of fabricating integrated circuits, defects will and do occur; the larger (and therefore more complicated) the circuit, the greater the chance of a defect. Integrated circuits are rigorously tested and the flawed ones discarded, thus operational circuits bear the cost of the defective ones as well. The eventual cheapness of amorphous computing devices comes from the fact that they are simpler, requiring less silicon, lessening the probability any given chip is defective, and thereby decreasing our overall cost. The non-monetary cost we pay is in computing nodes that are dead-on-arrival, or will fail some day due to initially non-crippling defects that eventually cause a failure.

1.1.1 Why Bother?

Of course, all of this begs the question, what benefit do we derive from this new mode of computing? Just because it's now becoming economically feasible to make hundreds, thousands, or millions of not-too-powerful computing nodes and scatter them about like grains of sand doesn't mean that we should. After all, the sea turtles and their favoring quantity hasn't enabled them to become the dominant species on Earth. However, there are several good reasons to add amorphous computing to our tool belts, although it will not replace more traditional modes of computing.

- It's cheaper to produce a lot of the same thing, even if they do more than you need, than it is to produce a few things that do only what you need.

- You can put amorphous computers places you couldn't normally put computers. Embed them in cement or hide them behind drywall. If some of them die, you're still fine. However, if you put all your eggs in one basket with a traditional computer, you're going to have to do a lot of remodeling.

- Amorphous computing can deal with harsh environments. As computing devices are introduced into our everyday life, they will be leaving the protected haven of the computer case, and subjected to real-world wear and tear. Computers will be woven into our clothing at some point; spills, bumps, and washing are the harsh realities that clothes face. Amorphous computing provides a technical solution to deal with this that does not involve trying to change human nature.

- The logical extreme of amorphous computing, nanocomputing, will arrive with nanotechnology some day. Nanocomputing is also the logical extreme of parallel processing, which becomes more important as time goes on and the physical limits of serial computation are approached.

Although amorphous computers may some day reach a scale when they are as small as a grain of sand, or can not be seen by the naked human eye, the devices in the real world are not there yet. The lower-bound on implementations on the street hover around the size of a bottle cap. This is in many ways a good thing, as our computing devices are then about the size of a reasonable power source that can power them for an interval of weeks to a year, depending on power usage. Computing devices with power sources can then be scattered throughout any environment without much concern for placement, save for the need of the devices to be placed within reliable radio range of each other. Much of the current size constraints have to do with the fact that the implementations are not yet integrated on a single chip, and thus require a circuit board and multiple components. The good news is that recent research results have produced single-chip solutions, allowing a dramatic decrease in size. The bad news is that we don't have ways to parasitically power those chips with heat from their environments. As a result, the battery defines the form factor. Alternatively, the miniscule devices can be somehow connected to a specialized substrate that provides them with power, but effectively limits the conditions in which the devices can be used. As such, the current state of the art represents a form-factor that will last long into the time of the sand-grain form-factor, by virtue of the domination of the power source. This is also true because the multi-chip architecture allows for specialization via expansion modules that connect to a data-bus or set of I/O pins, rather than requiring the production of a new, specialized single-chip implementation.

1.1.2 Current Applications

The primary application of amorphous computing at this time, and only clear win thus far, is what is known as "sensor fusion", a.k.a. sensor networks. Data gathering has a few key ingredients: the sensors (as many as possible), a way to get the data from the sensors to the people who need it (and their computers), and a way to maintain the infrastructure in the face of hardware failures. Let's look at this from both an amorphous computing and traditional computing perspective.

Amorphous computing devices are naturally suited to the problem. For some common tasks, such as temperature sensing, it is possible that computing nodes will already have sensors. In other cases, both digital and analog sensors can easily be integrated into the current-day, battery-scale computing devices trivially and with minimal cost. For example, the University of California Berkeley's lines of amorphous computing "motes" can speak SPI, a digital serial-bus protocol, and also possess analog-to-digital converters, allowing practically any electronic sensing chip to be integrated with no further hardware required. In order to communicate the data to a base-station, the nodes can use their radios to communicate amongst themselves, aggregating the data, and moving it to the intended destination. Because the nodes communicate locally amongst themselves, rather than broadcasting over great distances, power usage and the chance of collisions is greatly reduced. The failure of a sensor prevents a node from gathering further data, but allows it to continue routing messages as needed. The failure of a computing node removes a data point as well as a relay station from the network, but this only becomes a real problem when a massive number of nodes fail, potentially altering the network topology so that the network is split into two or more pieces.

Let's now examine a traditional approach to the problem. In this case, we'll use a number of computers with sensors, but with at least an order of magnitude fewer computers than we would have amorphous computing nodes. Our computers could be something as big and powerful as a state-of-the-art desktop computer, or as small and relatively weak as a high-end embedded device; what really matters is the computing model we're using. When it comes to sensors, we have two real possibilities. The first is to try and mimic the amorphous computing setup by having many sensors geographically distributed over an area, and running cables from our computing device to the sensors. If we follow this approach, we are probably going to require additional hardware to allow the computer to retrieve the data from the sensors. Alternatively, we could give each computer higher quality or more detailed sensors; if we're going to have an expensive piece of hardware attached to the sensor, the sensor might as well be expensive too. This additional precision might make up for having fewer data points. As for connecting our computers together, we can either go wired, or wireless. Each computer will communicate wirelessly via a global communications channel in a peer-to-peer fashion, with one computer being connected to the base-station. Unfortunately, our traditional computing configuration is much more vulnerable to failure than the amorphous computing approach. If a sensor fails, we're okay; it's just a data-point. However, if a computer fails, we lose an entire set of data-points (or a single very-important data point) in one instant. If we went wireless, we're safe from network issues. However, a wired implementation could lead to massive problems or massive expense. If the computers are daisy-chained, then a single computer or network-link failure can segment the network. If the computers are connected in a star-configuration, with a central switch, then we also have a central point of massive failure. Clearly, the traditional approach is more vulnerable to failures, and potentially more expensive. The more we do to make our approach less vulnerable to failure, the more it resembles a more expensive version of the amorphous computing approach.

1.1.3 Programming Model

Perhaps the greatest complication with amorphous computing is the programming model. There are many factors that come together to make programming an amorphous computer non-trivial.

- As previously noted, we have serious hardware limitations, which place limitations on what a single computing node can do. This means that if you want to do anything sizable, you can't do it all on one node.
- We can't depend on a node not to die on us.
- Network topology can't be ignored. We don't have some magical high-speed network connecting every computing node with near-constant time access. If two nodes want to talk and they can't hear each other, the message must be sent through other nodes, slowly, and contending with other traffic; worse yet, there's no implicit guarantee it will get where it's going. Also, if two nodes can hear each other, that means that if something other than random failure takes one node out, there's a good chance it'll take the other node out too.
- Emergent behavior is not always intuitive. Cellular automata like the game of life provide an excellent example of this.
- Debugging is hard. Simulation is in many ways the only viable technique to debug, but generally oversimplifies its model of the real world when it comes to message broadcast and receipt so much as to be useless. Even with a perfect simulator, it's still hard to know whether something has actually gone wrong or not unless the developer extensively uses design-by-contract or some other method of specifying intended invariants; this is not common.
- Parallel programming is hard. Most programmers louse it up in better situations than these (i.e., shared memory).
- Procedural/imperative programming does not automatically parallelize all that well. Most programmers use procedural languages.

When it comes to running programs on our amorphous computing substrate, there are two types of programs. The first type of program needs to know about the underlying network topology and its relationship to the real world. The second type of program doesn't care about those issues and wishes it was running on a traditional computer. What looking at that list of factors does is convince us that we need two separate languages to effectively program an amorphous computer. The first, lower level language, is designed for scalability to deal with the serious hardware limitations, and to allow us to easily provide the mechanisms to deal with the rest of the factors, without complicating programs semantically. The second language is to be implemented on top of the first language, shielding programmers from the intricacies of the platform that they do not need to deal with. This thesis deals with the specification and implementation of the lower level language, as well as providing the fundamental building blocks in that language which are required to establish the second language.

1.1.4 Viral Computing

Viral computing is a model of execution where code is king, capable of self-induced replication and mobility. This stands in contrast to the traditional execution model, where although code calls the shots, data is king. The traditional model is like a volleyball game; each half of the court is a computer/execution environment, each player a program, and the volleyball is data. It is trivial for data to be moved from computer to computer, that's the way things were meant to be. However, when it comes to moving a program, that's a different story; it's not a natural outgrowth of the model. Under a viral computing model, the players are eliminated from the game, and the volleyballs are self-propelled. Rather than requiring programs to move data about, code carries its own tightly coupled data from computer to computer. It is a more active model of computation, and in many cases, an easier way to express programs. Instead of saying, "If I receive this message, I process it and then I send this message", we now say "I arrive here, I do this, then I go there." In the first case, the I is a static piece of code, standing still and moving data; in the second case, the

I is an active capsule of code, moving itself about the computing substrates presented to it.

There are several additional properties that flow from the central idea of viral computing. For one, when introduced to an amorphous computing substrate and the inherent unreliability of broadcast communications, we must introduce the strategy of quantity. This strategy is inherent in real-world, biological viruses. If a virus manages to make its way into a living cell, it uses the machinery of the cell to make many copies of itself, then sends those new copies out into the world once more. Not all of the new copies will survive, so we cannot guarantee that any or all of them will make it to the next cell; we only know they survived if and when they do. This means that the most extreme implementations of a viral computing model won't provide reliable movement among computing nodes unless the viral capsules somehow accomplish it themselves. To do otherwise would be to create a hybrid existence with code beneath the capsule level playing a more active role.

The next key idea is that for viral computing to be successful, it must refrain from the excesses of exponential growth for its own good and the good of the computing substrate. Real RNA and DNA viruses cannibalize the cells they inhabit for their reproductive purposes. Computer worms and viruses also frequently fail to rein in their behavior, saturating their processing and communication substrates in a metaphorical blaze of glory before they are wiped out. The result is that our viruses or computing substrate must prevent this kind of behavior. Unless your goal is an artistic statement, there is little merit in having viral capsules overwhelming thousands of computing nodes to wage a ceaseless war.

1.2 Overview

Our goal is to make it eventually possible to write serial programs that are not aware of the underlying infrastructure or limitations of the amorphous computing substrate. The compiled programs would possess the required properties to resiliently execute in the face of random and deterministic failures of the computing substrate. The hypothetical language that supports this behavior we dub RSEAM, for Resilient Serial Execution on Amorphous Machines. To achieve this goal, we introduce the RSEAM Goo-Like Language, RGLL, that provides a language well suited to the problem domain of an amorphous computing, and provides the functionality required to eventually implement RSEAM. Although a full implementation of RSEAM is beyond the scope of this paper, we will describe the mechanisms required to enable an implementation of RSEAM to accomplish its goals, and how to go about implementing those mechanisms on RGLL.

We adopt a viral computing model as the basis for our language. The rationale is that this model of execution is better suited to an amorphous computing substrate than a traditional model of execution. Regardless of execution model, packetized data will be moving throughout the network. The viral computing model is better suited to dealing with this reality. We semantically attach the actions associated with the movement and processing of this data to the data itself. The alternative is a whisper-down-the-lane model of execution: "Johnny said this, so I should do this, and then tell Suzy this." The contrast is similar to the difference between dynamic and static languages. One of the great benefits of dynamic languages, at least as put forth by those who use them, is rapid development and clarity of implementation. The fact remains that a static implementation may edge out the dynamic implementation in terms of runtime speed and efficiency, but only after potentially longer development of a larger code base that had more bugs. This is our argument. By adopting a viral computing execution model, our language is better suited to the problems at hand. As we will also attempt to prove, this also imbues our language with additional scalability.

1.3 Related Work

1.3.1 Amorphous Computing

The Amorphous Computing group at MIT's Lab for Computer Science and Artificial Intelligence Lab has pioneered the field of Amorphous Computing [AAC$^+$00], and defined the fundamental operations for this model of computing. They have provided algorithms for broadcast, gradient formation, leader election (a.k.a. club formation) [CNW97], and coordinate formation [Nag99]. Their work is biologically inspired. The Amorphous Computing group has concentrated on the development of algorithms and languages for expressing structural growth on an amorphous computing substrate, rather than the development of general purpose programming languages. As such, there is not too much to directly compare against.

However, a paper of note, Amorphous Infrastructure for Language Implementation [BN02] describes the implementation of a limited Scheme implementation on an amorphous computing substrate. The implementation described shares many goals with RSEAM, including resiliency in the face of random and deterministic node failures, and potential leveraging of parallel execution. However, the implementation is also fundamentally different at several levels. First, the amorphous computing substrate is assumed to be comprised of much simpler nodes, with each node taking on the tasks of a single graph node, or acting as a very-constrained storage cell. This likens the implementation to a cross between compilation-to-silicon and biological computing. Secondly, the implementation appears to be trying to implement a virtual machine of sorts on top of the amorphous computing substrate. This means memory allocation, cons and cdr cells, the whole shebang.

1.3.2 Paintable Computing

Bill Butera's Paintable Computing simulation work at the MIT Media Lab [But02] introduces a hybrid viral computing/traditional computing execution model and several important applications based on the viral computing model. The viral computing aspect of his model is that programs are actually process fragments which are capable of self-locomotion throughout the network. They can replicate themselves and kill themselves off. Using this behavior, Butera implements a series of applications such as the establishment of multiple communication channels and resilient holistic image storage.

However, the implementation does not go all the way in terms of viral computing, and still has many traces of the traditional execution model. Data exchange between process fragments on the same node is accomplished via the “Home Page”, which is basically a shared scratch pad. All nodes also mirror copies of their neighbors' home pages, allowing for inter-node communication. Although process fragments carry internal state with them when they move, it is not their primary method of moving data. This deviation in their treatment of data means that the programming model cannot benefit from the I-as-data semantics, and must limit itself to the I-as-data-processor semantics. The execution model also deviates in that process fragment movement is reliable; a process fragment requests to be copied to another node, and is notified whether its request will be honored or not before it makes any decision about deleting itself or taking any other actions.

The result of these deviations is that although Butera's implementations undeniably conform to the viral computing paradigm at a high level, moving throughout the network of their own volition, and building on that functionality; it stops there. The underlying programming model is still the C programming language acting as a data processor. Although it remains to be seen whether this hybrid approach is better than an all-out viral computing implementation, it should be noted that Butera's thesis indicated that the home page data model was discovered to not be ideal during the implementation of the applications.

1.3.3 Berkeley Smart Dust

The University of California Berkeley has been one of the most prolific developers of software and hardware for sensor fusion in recent history.

Hardware: Motes

The WEBS Group at the University of California Berkeley has implemented several hardware platforms for "sensor fusion" purposes [CHB$^+$01]. The Berkeley computing nodes, knows as "Motes" or "Tinies", operate at a different scale than those conceived by the Amorphous Computing Group or Butera's Paintable Computing. The current generation of motes is relatively large in size; the standard form factor is defined by the two AA batteries that comprise their power source. However, a smaller form factor that is approximately the size of a quarter is available, and recent work on a very small single-chip implementation has apparently been successful. Communication is accomplished via single-channel radio broadcast, although this may change in the future.

Operating System: TinyOS

TinyOS is an operating system in an embedded computing sense; it provides the mechanisms that allow your program to interact with and control the underlying computing hardware, but is best thought of as a library rather than an operating system in the desktop computer sense. TinyOS provides its own language, NesC, and compiler, ncc [GLvB$^+$03], in order to provide syntactic support for the idioms it introduces. Ncc acts primarily as a preprocessor, allowing a new, cleaner syntax that replaces the previous release's numerous required macros laid on top of straight C.

Programming Paradigm

TinyOS breaks software down into modules defined by interfaces. A program is created by wiring modules together via their various interfaces. Interfaces are comprised of commands, tasks, and events. Commands are provided by an interface to its consumers to invoke. Tasks are low-priority, interruptible snippets of execution that are scheduled through a task queue, allowing deferred execution. Events are non-interruptible pieces of execution, triggered by the provider of an interface, that accordingly must be kept short in duration. The execution model is strictly traditional; code is static and immobile, and messages are passed purely as data between motes.

However, this is not necessarily a bad thing. Just as directly-accessible register machines seem to be the fundamental base case for processors, this model likewise is an important base-case for the platform. For example, if you go through the motions of implementing stack-based processors, you quickly find that for an efficient implementation you need all the operations required by a load/store machine. As such, it makes more sense to just implement a load/store machine, and then build on that if you want anything more. The same is true in this case. Our language, RGLL, and its viral execution model can be implemented in the TinyOS paradigm, and in fact were designed to do so. In contrast, it is impossible to implement the TinyOS model on top of RGLL. However, it is our argument that using RGLL allows a programmer to more efficiently develop a program than using the TinyOS model. Again, the argument is roughly parallel to dynamic languages versus static languages.

1.3.4 Fault-Tolerant Computing

Fault-tolerant computing is found in industries where neither down time nor data-loss is acceptable. Although no one actually desires down time or data-loss, in practice, very few people are willing to pay the price to banish them. The battle cry of fault-tolerant computing is "redundancy", and specialized, expensive hardware is required to achieve this when high throughput is required. In lower performance cases and/or tasks with more parallelizable transactions, off-the-shelf hardware used en masse can accomplish the same goals for a lower, but still non-trivial, cost.

1.3.5 Comparison

Butera's work describes an excellent model of mobile computation, as well as several important algorithmic implementations of common amorphous computing algorithms in that model. Likewise, TinyOS provides a compact, efficient OS which allows the construction of programs that operate in a message passing model. Although both of these projects are excellent implementations that work well, they fail to deal with the problem of building programs that are larger than a single computing node.

In the Paintable Computing model, to contend with such a task, you would break the problem down into multiple relatively monolithic program fragments and work out a protocol for their interaction. Likewise, in TinyOS, implementing functionality too large for a single mote requires slicing the desired functionality into smaller modules and distributing them to various motes. This problem is exacerbated by the lack of mobile code. The problem with this common programming model is that it does not scale, particularly in the case of TinyOS. What is required is a program language and programming model that leverage the capabilities of both systems, while making it easier to build larger, more resilient systems.

Chapter 2

RGLL

2.1 Language

The RSEAM language, RGLL, is currently derived in both syntax and semantics from Jonathan Bachrach's Goo language [Bac03], hence its expanded name, the RSEAM Goo-Like Language. It uses the same keywords and special forms as Goo whenever possible, however, it is not yet sufficiently featureful to be considered even a proper subset of the language. It is best to think of the language as a simple embedded Scheme virtual machine wearing the syntax mask of Goo.

2.1.1 Types and Variables

RGLL has a relatively limited type support at the present moment. Integers, booleans, and lists are the currently supported types. All variables are statically typed for efficiency; once a variable is declared, its type is set in stone. If provided by the underlying hardware or required sufficiently to justify software emulation, floating-point type support would be an excellent addition. For simplicity, I have restricted my implementation to just those types that are required for the mechanics of building an RSEAM-capable implementation.

2.1.2 Syntax

RGLL uses a prefix syntax, and has minimal syntactic sugar/deviations from pure prefix form. Integers and boolean values are self-evaluating, with #t and #f conveying true and false, respectively.

Syntactic Deviations:

- *'abc* expands to *(quote abc)*, where *quote* is a special form described below.
- *a:b* is a nested reference to the *b* member of *a*. RGLL proper does not provide nested structures, but the functionality is provided for use when capsules enter the picture. In those cases, the syntax provides a way to access multiple capsules of the same type when code needs to refer to more than one capsule of the same class, or as a namespace convenience when referring to multiple parents. Nested references may be arbitrarily deep.
- *@expr* is used in the definition of functors, and is explained in more depth later. It acts like an unquote inside of a quasiquote, evaluating the subsequent expression in the current context when a functor is invoked, much like macro expansion in Goo.

2.1.3 Special Forms

defconst *constant-name constant-value*

Sets the given *constant-name* to *constant-value*. This is truly a constant which is resolved at compile-time, and is never made visible in any run-time environment.

seq *body...*

Executes all of the expressions in the body. The return value is the value returned by the last executed expression, or #f if the body is empty.

quote *symbol*

Quotes the given symbol. Symbols are resolved to integer ids at compile time.

if *test true-case false-case*

Executes the expression *test*, subsequently executing *true-case* if the result was non-false, or *false-case* if the result was false.

when *test body...*

Executes the expression *test*. If non-false, the statements in the body are executed, and the result of the final expression is returned. If false, #f is returned.

unless *test body...*

Executes the expression *test*. If false, the statements in the body are executed, and the result of the final expression is returned. If non-false, #f is returned.

case *comparison-expr (match1 expressions...) (match2 expressions...) ...*

Executes *comparison-expr*, and compares it sequentially with the first value in the lists that follow. The first list whose match-value matches has its list of expressions executed, with the result of the final expression being the returned value. If there is no match, *case* returns #f. There is currently no support for an else clause, but it would be a good idea.

set *binding-name expr*

Locates the binding named *binding-name* in the current environment chain, and sets its value to the result of executing *expr*. If no binding is found, one is implicitly created in the local environment.

def *binding-name expr*

Creates a new binding named *binding-name*, storing the result of executing the expression *expr* in the local environment using the new binding.

and *expr1 expr2 ...*

Short-circuiting boolean logic. Expressions are evaluated in order. If any expression returns #f, then evaluation terminates and #f is returned. If all expressions return non-#f, then the value of the final expression is returned.

or *expr1 expr2 ...*

Short-circuiting boolean logic. Expressions are evaluated in order. If any expression returns a non-#f value, then evaluation terminates and the value is returned. If none of the expressions evaluate to non-#f, then the return value is #f.

for *((binding-name1 list1) (binding-name2 list2) ...) body...*

Provides (parallel) iteration over one or more lists. Although the intention is that all lists will be of the same size, the first list drives the actual iteration. Before each iteration, the bindings in each (binding-name list) tuple are set to an indirect-reference to the given list element. The use of an indirect reference allows reads of the value to return the current value, but a *set* will overwrite the current value in the list. During each iteration, all of the expressions in *body* are executed. The result of the execution of the body is ignored, and *for* returns #f regardless.

find *((binding-name1 list1) (binding-name2 list2) ...) body...*

Find is a specialized version of *for* that returns the first non-false result of executing the body of the loop. If no iteration returns a non-false value, then the result of *find* is #f.

2.1.4 System Functions

random *high-non-inclusive*

Returns a random number between 0 and *high-non-inclusive*, non-inclusive.

random-range *low-inclusive high-inclusive*

Returns a random number between *low-inclusive* and *high-inclusive*, inclusive.

list *items...*

Returns a list containing the result of evaluating the contained sub-expressions, if any. For example, (list 1 2 3), (list), and (list #t #t #f) are all valid.

mem? *list value*

Returns a boolean indicating whether the given value is present in the list.

len *list*

Returns an integer whose value is the number of items in the list

add! *list value*

Adds the given value to the end of the given list, mutating the list. Returns #t.

del! *list value*

Removes the given value from the given list, mutating the list. Returns #t. Note that the semantics of this function are identical to Goo's del-vals!, rather than Goo's del!.

zap! *list*

Empties the given list (mutating). Returns #t.

elt *list index*

Returns the value in the list with the given zero-based index.

intersect! *list1 list2*

Mutates list1 to contain the intersection of the values in list1 and list2. Returns #t when the result is non-empty, and #f when empty.

intersect *list1 list2*

Returns a list containing the intersection of the values in list1 and list2.

union! *list1 list2*

Mutates list1 to contain the union of the elements in list1 and list2. If both lists contain elements that have equivalent values (using =), that value will be represented only once in the resulting list. The order of the resulting elements is equivalent to appending the result of (diff list2 list1) to list1. Returns #t when the mutated list1 is non-empty, and #f when empty.

union *list1 list2*

Returns a list containing the union of the elements in list1 and list2. If both lists contain elements that have equivalent values, that value will be represented only once in the resulting list. The order of the resulting elements is equivalent to appending the result of (diff list2 list1) to list1.

diff *list1 list2*

Returns a new list containing all the elements of list1 that are not present in list2. Given our pass-by-value semantics, "present" in this case refers to the presence of an element with the same value.

= a b

Tests a and b for equality of value. List equivalency is set equivalency; all of the elements in a must be in b, and all the elements in b must be in a.

+, -, /, * a b ...

Performs the expected arithmetic operations on the arguments.

2.2 A Running Example

To aid in the process of explaining the various language features and motivate their inclusion in the language, we will be using a simple tracking system [NSB03] drawn from our research as a running example. We will assume that we have several hundred computing nodes distributed on a 2d plane throughout some space. These nodes will be distributed with sufficient density so that every node has several other nodes within its radius of successful radio transmission. Any physical obstacles on this plane, such as walls, have no noticeable effect on the radios used for communication, keeping our topology simple. Some node in the network is a base-station, connected to a desktop-class computer which is our interaction point with the real world. All of the nodes have unique ids assigned to them. We establish a coordinate system by having the physical locations of several nodes and their unique ids provided to us. The locations are gathered by having graduate students with GPS transceivers wander around and jot down the locations of several nodes, assuming the ids are printed on their exterior.

This information is relayed to the base-station host computer. There exists an object to be tracked, and this object has a computing node attached to it. Our goal is to track the object throughout the network.

In order to implement this example, we will use several key algorithms from amorphous computing. We describe the algorithms here, and then will describe how they come together to implement our example.

Flood Broadcast

The concept of a flood broadcast is relatively simple; we have a message that we want to get out to every node in a group of nodes, but we don't want to build up a complicated hierarchy to do so. We want to ensure that every node that could possibly hear the message hears the message. Furthermore, we want to ensure that this is done in a finite amount of time; we don't want our messages rattling around for all eternity. A flood broadcast achieves this by the following two rules:

1. If we hear a flood message and we have not already heard it, process it and then rebroadcast it.

2. If we hear a flood message and we have already heard it, ignore it. For the time being we will not address the issue of how long we remember that we heard the message.

Gradient Formation

Gradient formation assigns a scalar value to every node in a group of nodes that indicates its distance from the origin node of the gradient. In practice, the distance is the number of hops, but when hardware allows reliable signal strength measurement, the units may be something other than the number of hops. The goals of the algorithm are similar to those of the flood broadcast; we want to have all nodes that could conceivably be involved in the gradient to be involved, we want the process to take a finite amount of time, and we want it to terminate. Gradient formation is fundamentally a broadcast operation, and so operates regardless of any established hierarchy. The pseudo-code for gradient formation goes like this:

Base Case: The origin node sends out a gradient message with a distance of 0.

Inductive Case:

1. If we hear a gradient message, and we have not heard a gradient from the given origin before OR using its value would lower our known distance to the origin, use its value and then broadcast our revised distance estimate.

2. If we hear a gradient message, but using its value would leave our known distance the same as it was before, or increase it, ignore the message.

Example Mechanics

A quick sketch of the operation of our running example follows.

- The base-station floods commands to each of the "anchor" nodes for which we have a physical location, telling them each to emit a gradient.

- Each anchor emits a gradient, with the intended goal that every computing node will know its distance in hops from each of the anchor nodes.

- The base station emits its own gradient so that the nodes in the network can efficiently send messages to the base-station via gradient descent.

- The object to be tracked enters the network. It periodically broadcasts a message announcing its presence to the network. All of the nodes that hear the message send messages to the base-station, letting it know. By knowing all of the nodes that heard the broadcast, and knowing the approximate location of those nodes, we can calculate an approximate location for the target for each of the broadcasts. The result is that we can track the object in question throughout our network. For our purposes, we will assume that the node attached to the object assigns a unique ID to each broadcast it makes. The alternative technique to disambiguate the broadcasts would be time synchronization, which

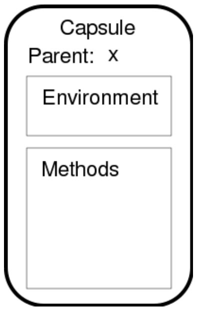


Figure 2-1: The basic code capsule concept.

would result in a more realistic implementation, and one more conducive to more realistic sensor-based tracking. However, this would add unneeded complexity to our example, so we'll stick with the simplification.

2.3 Capsules

Our computing nodes deal exclusively in code capsules. Every message they send is a code capsule; every message they receive is a code capsule. Code capsules are indivisible; if you have anything less than a whole code capsule, all you have is garbage data. Computing nodes can store as many capsules as they have space for. But what makes a code capsule?

- A name. In source code, this is a unique symbol. In an actual compiled capsule, this is a unique integer id. This name is like a class name in C++; it is not an instance name. Uniqueness of symbols is a programmer burden; don't make two capsules and name them *foo* unless you want an error. If new capsules are programmatically generated, new unique symbols will be generated for them; if user access to the symbols is required, they will be provided by compile-time constants. The compiler is responsible for assuring that unique integer ids are used for deployment. In the trivial case, only one compiled RGLL program is introduced into the network at a time, and translating the unique symbols to unique integer ids suffices. If more than one program is to be introduced to the network at a time, then the compiler and/or user are responsible for keeping the id ranges from overlapping.

- A list of the names of environmental parent capsules. The list may be empty. Like the capsule name, the contents of the list are symbols in source and unique integer ids when compiled. They identify capsules who we should link our capsule environment to, with the named capsules becoming parents of our environment. By having multiple capsules with the same parent, we can effectively establish shared memory. The environmental links are transitive, so if C has B as a parent, and B has A as a parent, then C's environment will link to B's and then to A's. Cycles are prohibited.

- A capsule environment. In source code, this lists all of the bindings and their default values. The default values implicitly define the types of the variables. In a binary capsule, the values take the form of a packed structure which instructions explicitly reference.

- Capsule methods. In source code, methods are defined by giving them a symbolic name, and then providing a list of instructions. In actual capsules, the method names resolve down to integer ids, and the code is compiled into bytecodes. It is important to note that methods and their implementation are not affected by an environment parent, if specified. All the code that a capsule uses, it carries around with itself. Methods may optionally specify a list of arguments. If arguments are specified, they must be present when the method is called. Arguments are assigned to variable names in the order which the arguments were defined; there is no concept of keyword arguments for capsule methods.

2.3.1 Capsule Standards

Experience derived from RGLL's predecessor prototyping language and case-analyses of the tasks that would be required of RSEAM showed that the code logic that would be placed in a code capsule could easily be broken down into four categories:

- Code that is executed when a capsule arrives on a node for the first time (init code).
- Code that is executed periodically (tick code).
- Code that is executed when a capsule arrives on a node and encounters another instance of itself (collision code).
- Code that is executed to prepare a capsule being broadcast to other nodes (packup code).

Butera's paintable computing implementation also exposes a number of standard functions to the run-time layer while allowing for additional functions that are not part of that exposed interface. The standard actions defined are "Install", "De-Install", "Transfer-granted", "Transfer-denied", and "Update". The "install" method is roughly analogous to RGLL's *init* method, while "transfer-granted" is similar to *packup*, except that we have no guarantee that our broadcast is going to go anywhere at all. Because the paintable computing model uses reliable code movement, it has no equivalent to our *collide* method, which is probably the most useful of our standard methods. "De-Install" and "Transfer-denied" have no analog in our model. The "Update" function is not a perfect match with our *tick* function. "Update" is the method by which the operating system bestows a time slice for execution on a process fragment, and if the fragment is going to do anything, it had better do it then. Although we try and centralize our periodic behavior in our *tick* method, we can schedule multiple methods on a capsule for execution, allowing us more flexibility. Additionally, the *collision* method allows us to easily implement complex functionality without constant polling.

Some implementations of amorphous computing algorithms assume that they can "hear" a collision and infer something from that collision, but such an assumption does not fare as well in the real-world and in cases where more than one type of message may be sent.

init semantics

The *init* method is executed when a capsule first successfully arrives on a node. Nodes that are starting up treat capsules that are supposed to be loaded on startup as if they came from another node; as such, we are assured if a capsule is present on a node, it has been init'ed. If a capsule arrives on a node, and an equivalent capsule is already present, the collision function of the incoming capsule is used to decide which of the capsules gets to stay. If the incoming capsule wins, then its init function is invoked and it stays. If the already-existing capsule is picked over the incoming capsule, then the incoming capsule's init function is never executed. In this case, the already-existing capsule's init function has already been executed previously.

If periodic behavior is required of the capsule, then the init function should start the behavior by scheduling a tick.

tick semantics

The *tick* method is executed after n ticks as the result of calling *schedule-me n*. Because scheduling only schedules a capsule's *tick* function for execution once, the *tick* method is responsible for re-scheduling itself if its behavior should occur periodically.

In our execution model, we assume that all computing nodes have internal timers generating tick events at approximately the same frequency, although phase need not be synchronized. As such, algorithms should assume that 4 ticks on any node in a group of nodes should last approximately the same amount of time, but should not assume that the ticks will occur at the same time. The use of rounds in our simulations is a simplification.

Capsules should not attempt to do everything every tick just because they can. In more realistic simulation conditions, the number of ticks per second would be made equal to the number of messages that can be transmitted by the simulated hardware in a second. In such a situation, the use of ticks is primarily to help eliminate message collisions, as well as perform behaviors periodically.

If and when real-world timing is required, scheduling functions will be provided that take time intervals in terms of real-world units, independent of simulation-scaling artifacts.

collision semantics

As previously noted, a successful viral computing execution model won't permit exponential growth to occur; we derive no benefit from filling up all available memory with copies of the same capsule. For this reason, we allow only one instance of a capsule per computing node. Of course, it's no good if we just throw away additional copies of a capsule when we receive them; the new capsule might have some useful information. Instead, we allow the capsules to decide who gets to stay, while also potentially reconciling differences in state.

The *collision* function is executed on an incoming capsule that is equivalent to a capsule that is already resident on the node. The function is responsible for determining which capsule (and its respective state) should be the resident capsule on the node, and which should be discarded. In order to determine which capsule stays, the function needs to be able to investigate the state of both capsules. The state is provided through the *resident* and *incoming* arguments. Using the nested *a:b* syntax, the state variables of each capsule can be examined. For example, if a capsule has a variable "foo", it would be accessed as *resident:foo* and *incoming:foo*. Variables can also be mutated, allowing for the reconciliation of data between the two capsules. The *collision* function indicates which capsule should be resident by returning *resident* or *incoming* as appropriate, remembering that the function is executed on the incoming capsule.

If no collision method is defined, then the behavior defaults to selecting *incoming*.

packup semantics

The *packup* method is invoked on a capsule that is about to be broadcast. This occurs after its state has been 'cloned' from the copy that is staying, but before the transmission actually occurs. This allows a capsule to make changes to an outgoing capsule without erroneously affecting the state of the resident capsule.

2.3.2 Capsule Syntax

A code capsule is defined as follows:

```
(capsule CAPSULE-NAME (PARENT-CAPSULE-NAME1 PARENT-CAPSULE-NAME2 ...)
   ((VARIABLE-NAME1 VARIABLE1-INITIAL-VALUE <NAME1-NAMING?>)
    (VARIABLE-NAME2 VARIABLE2-INITIAL-VALUE <NAME2-NAMING?>)
    ...)

   (METHOD-NAME1 (METHOD1-ARGS...)
      METHOD1-CODE...)

   (METHOD-NAME2 (METHOD2-ARGS...)
      METHOD2-CODE...)
 )
```

The naming attributes of state definition are optional, and by default assumed to be #f. Variables that are marked as "naming" comprise part of the capsule's name for the purposes of uniqueness. Only one instance of a capsule can exist on a computing node at a time. By allowing some of the capsule's name to be derived from its state, we allow something akin to named partial-evaluation. This is important for any situation where we have a real-world function *f* that takes one or more parameters that we are trying to implement in RGLL/RSEAM. Clearly, evaluation of *f(1)* is quite different from evaluation of *f(2)*. The use of naming variables allows this distinction to be obvious to our implementation at a very low level, and supports the existence of *f(1)* and *f(2)* simultaneously on a single node. However, multiple instances of *f(1)* may still not exist on the same node.

If two or methods with the same name are defined inside the capsule, only the first definition is used; subsequent definitions are discarded.

Pseudo Example:

```
(capsule hello-world ()
   ((a-number  1)
    (a-boolean #t)
    (a-list-of-numbers (list 1 2 3)))

    (init ()
      ...)

    (tick ()
      ...))
```

2.3.3 Capsule equivalence

Two capsules are said to be equivalent when their globally unique Ids are equivalent and any state variables that are marked as part of their names are also equivalent.

2.3.4 Capsule-Level Functions

When referring to capsules, it is important to realize that at the binary level, capsules will always be referred to by their unique integer id. From a programming perspective, however, there are several ways you may reference them, all of which are equivalent. First, you may reference a capsule by its symbolic name, which resolves to its unique integer id at compile time. Secondly, you may reference a capsule through a constant that resolves to its unique integer id at compile time. Lastly, you may reference the capsule via an integer variable. The integer variable derives valid unique id values either from symbolic capsule names that resolved to integers on compilation, or from functions that return valid integer ids. It is important to note once more that there will only ever be one capsule on a node that has a given unique integer id.

The following functions are introduced to deal with capsules.

call *method-name arguments...*

Executes the function on this capsule with the given name. The *method-name* must be a symbol. All arguments defined by the method must be present. Execution is synchronous, and the return value of the call is the result of evaluating the given method.

call-other *capsule method-name arguments...*

Executes the given method on the given capsule on the current computing node, assuming the the other capsule is present. *Method-name* must be a symbol. All arguments defined by the method must be present. Execution is asynchronous, guaranteed only to take place after the current code block finishes executing. Returns a boolean indicating whether or not an instance of the given capsule exists on the mote, and therefore whether the method will be invoked or not.

broadcast *capsule*

Places capsule on the queue of capsules to be sent. Returns #t.

broadcast-me

Places the current capsule with the current state on the queue of capsules to be sent.

schedule-me *ticks*

Schedules execution of the current capsule's associated tick routine in *ticks* ticks. Returns #t.

schedule *method-name ticks*

Schedules execution of the current capsule's given *method-name* in *ticks* ticks. Note that *method-name* is assumed to be a symbol. Returns #t.

me-scheduled?

Returns a boolean indicating whether or not the current capsule's *tick* method is scheduled to be executed.

schedule-other *capsule ticks*

Schedules execution of the given capsule's associated tick routine in *ticks* ticks. Returns #t if the capsule exists on this node and therefore scheduling has succeeded, and #f otherwise.

capsule-present? *capsule-name*

Returns a boolean indicating whether or not the given capsule is present on the current node.

purge-me

Flags the current capsule for removal once the current code completes execution.

node-id

Returns an integer value that uniquely identifies the current node.

2.3.5 Execution Model

The number of capsules stored on a computing node is limited only by the memory resources of the node. There is no limitation on the number of capsules that can be "active" at any given time. Capsules are responsible for scheduling their own method execution via the various scheduling functions when they arrive and their *init* method is executed. Capsule method execution is atomic. The only time a method will not run to completion before another capsule method is invoked is if a capsule method invokes other methods on the same capsule during its execution, and in such a case, that is the desired behavior. This helps eliminate the various race-conditions that arise in a lower level implementation. Capsules effectively execute in a round-robin fashion, taking turns having their methods executed.

Scheduling via ticks does not provide tight time guarantees. It merely ensures that your code won't be executed before that number of ticks have elapsed. If a node is swamped with ticks to be processed, the rate of ticks effectively slows down; tick invocations will not be dropped to try and maintain real-time behaviors.

Nodes that do not have any future execution scheduled are not assumed to be dead, and will not automatically be "garbage collected". There are many valid reasons for a capsule to hang around indefinitely without consuming any resources. However, to avoid the worst-case scenario of a node completely filling up with capsules, an implementation may opt to discard the nodes that have not been "used" for the longest period of time. In this case "use" is defined as having one of its methods executed, or having one of the methods of its environmental descendents executed. In other words, nodes who are active, or whose existence has allowed other nodes to be active, would not be purged.

2.3.6 Example: Introducing the Tracking Players

Let's sketch out simple implementations of the various capsules we'll need.

Command Flooding

In order to notify each anchor node that it is an anchor and that it should emit a gradient, we flood commands throughout the network from the base station. The following code provides this functionality.

```
(defconst REBROADCAST_LOW  2)
(defconst REBROADCAST_HIGH 5)

(capsule flood ()
   ((target-node 0))

   (init ()
     (schedule-me (random-range REBROADCAST_LOW
                                REBROADCAST_HIGH))
     (when (= target-node (node-id))
      )) ;; This is where we will initiate the gradient.

   (tick ()
     (broadcast-me))

   (collide (resident incoming)
     ;; If we're already present, favor resident to stop the flood.
     resident))
```

Our *init* method schedules our tick function to be called a random number of ticks in the future. The use of *random-range* is excessive for our simplified goals and assumed world, but would be more important in an environment where we need to be concerned about message collisions. The *init* method holds our payload, invoking it only if the current computing node is the intended recipient of the message. Regardless, the *tick* method broadcasts the message. The *collide* method ensures that once we have received the *flood* capsule once, we will not broadcast it a second time, nor will we execute our payload a second time. By always returning *resident*, *collide* ensures that the resident capsule always stays resident, and that a new incoming copy of the *flood* capsule will be ignored, along with its *init* and (transitively) *tick* functions.

Gradient Propagation

The anchor nodes initiate a gradient propagation when told to do so by the base-station. Also, the base-station emits a gradient so that packets directed to the base-station can simply follow the gradient back to the base-station. The following code provides this functionality.

```
(defconst SEND_DELAY_LOW 2)
(defconst SEND_DELAY_HIGH 5)

(capsule gradient ()
   ;; State
   ((grad-val 0))

   (init ()
     ;; If we are here, this means we are the first capsule of our kind on
     ;;  this mote, or we were less than the previous value.  Either way,
     ;;  this means that we should re-transmit ourself after a suitable
     ;;  anti-collision delay
     (schedule-me (random-range SEND_DELAY_LOW SEND_DELAY_HIGH)))

   (tick ()
     (broadcast-me))

   (collide (resident incoming)
     (if (< incoming:grad-val resident:grad-val)
         incoming
         resident))

   (packup ()
     (set grad-val (+ grad-val 1))))
```

The only state we maintain is the number of hops from the origin of the gradient. As noted in the comments, there are actually two ways this value should be initialized, depending on the implementation path taken. We are assuming that only the origin node has this capsule at the beginning of execution. If, instead, every node started out with the *gradient* capsule present, we would want the default value to be the maximum integer representable; this is in-keeping with the practice of taking the minimum of our current distance estimate and any new estimate we hear. By convention, we define that the distance estimate is updated on transmission rather than on receipt. For our purposes, either technique works. In a situation where signal strength is analyzed, however, the estimate would need to be updated on the receiving node.

As in the case of the flood broadcast, our *init* method schedules a tick to be made in the future and our *tick* method rebroadcasts us. The *collide* method provides our minimizing logic. If a node receives the *gradient* capsule without having received one previously, the *collide* method does not even come into play, and the *init* method is invoked. If, however, we receive a new *gradient* capsule when one is already resident, the *collide* method discards it unless its distance estimate is less than our current distance estimate. The *packup* method increments our distance estimate before we broadcast the capsule to other nodes. As a reminder, the *packup* does not operate on the resident capsule, but instead operates on a copy that is actually what will be transmitted.

Gradient Descent

The last player is the capsule emitted by the node attached to the tracked object. This capsule performs gradient descent, following a gradient emitted by the base-station back to the base-station. The capsule only moves in the direction of lower gradient values, which leads it back to the base-station. In order for this to work, our gradient descent capsule must be able to interact with the emitted gradient, finding the *grad-val* associated with the gradient for the current node. However, this requires functionality that we haven't yet covered. We describe that functionality now.

2.3.7 IPC and Shared Memory

In order to accomplish anything non-trivial, our capsules must be able to communicate and work together. From the functionality we have presented so far, we can implement a kind of inter-node communication by having capsules broadcast themselves to other nodes that already possess an instance of themselves. When the capsule is received at the other node, the *collide* method provides the capsules with a way to reconcile their data. It's a form of primitive inter-node shared memory. We do not attempt to provide any abstraction that allows multiple nodes to share a memory space mapped among the nodes. To do so would violate our adopted precepts of viral computing; the substrate on which our code executes would be just as smart, if not smarter than our capsules themselves. It is also a question of abstraction; reliable shared memory conceptually exists at a higher level than RGLL. But perhaps the strongest argument is simply that it would be very hard to implement the functionality in a usable fashion in the face of unreliable transmission.

More than the ability for capsules of the same type (read: same unique ID) to communicate, it is important for different capsules to be able to exchange data and notify each other of events. Without this ability, it would be impossible to build up more complex functionality through the interaction of multiple capsules; RGLL programs would have their maximum complexity bounded by their maximum capsule size. We provide a novel method of providing intra-node shared memory, and a method of issuing cross-capsule function calls as a method of generating events. Together, these provide a usable foundation for intra-node inter-process communication, and the building blocks for creating your own inter-node shared memory infrastructure.

Intra-Node Shared Memory

Most lexically-scoped languages, can have their memory/variable infrastructure viewed as a hierarchy of environments that are linked together based on lexical nesting. When searching for the value of a variable/binding, we check the most-specific environment. If the variable isn't in that environment, we go to the current environment's parent. This continues until we run out of parents or we find the variable we're looking for. This hierarchy necessarily provides a form of shared memory. For example, in a C++ program, two classes defined in the same file will both see any top-level variables defined in the file. The potential problem is that since the hierarchy is implicitly defined by the lexical hierarchy that the code is defined in, it would be a rather painful process to use shared memory if that's how it had to be implemented. In fact, unless the language adopts a dynamic scoping model, that method prohibits any shared memory that is orthogonal to the way in which the code was written. The traditional approach is simply to allocate memory that can be seen by everyone who needs to see it. In a single process, this is accomplished by allocating memory on the heap. When spread among multiple-processes, operating-system functionality like System V shared memory is used. The problem with this approach is that it is not transparent. Someone must be responsible for allocating the memory, and someone must initialize it. This could cause everyone who wants to access the shared memory to carry around a lot of extra baggage.

Our solution is to transform the implicit environment hierarchy into an explicit hierarchy. This frees us from the limitations of lexical scoping, while also making the initialization semantics of shared memory quite clear. If two capsules want to have shared memory, then their "environmental ancestors" must overlap at some point. The initialization semantics are clear; the ancestor provides the default values. If an ancestor is not present, then its "environmental descendents" are implicitly disabled. This simplifies the logic as well, without any loss of generality. If you want your capsule to run, make sure its ancestors are present on the capsule. A mechanism for such capsule distribution is described later. When writing a capsule, you may specify as many environmental parents as you desire, including none. The only restriction is that the hierarchy may not contain loops. The rationale for providing multiple-environmental inheritance is intended for those capsules who need to consume data from disparate sources. Without multiple-environmental inheritance, it would not be possible for the consumer to see the multiple sources it needs without a massively convoluted, and potentially impossible, chain of environments that is tantamount to obfuscation.

It is important to note what this mechanism is not. This is not method inheritance. Capsules inherit no behavior from their environmental ancestors. This is also not a link that crosses node boundaries. When a capsule arrives on a node, a check is run to determine whether its environmental parents are present, whether the parents' parents are present, are so on, until we run out of parents. If all the ancestors are present, the capsule is enabled and the environmental links established. If the ancestors are not present, the capsule is disabled until the ancestors eventually arrive. To really drive this point home, think of having an environmental parent as having a membership card to a chain of gyms. Anywhere you go, you can get into the gym if there is one there, but it's clearly not going to be the same gym you were at in the last town.

Inter-Capsule, Intra-Node Asynchronous Method Calls

Although a shared memory infrastructure can accomplish many of our goals, we still need a method for inter-capsule events to be generated. In order to build up complex functionality in an abstract way, we need the ability to have loose-coupling between capsules; cross-capsule method calls provide this. Cross-capsule function calls are basically miniature messages sent between capsules, and their semantics are designed that way. The function calls are asynchronous. When you invoke the *call-other* function, it posts the method call to a queue. The method on the other capsule will only be executed after our currently executing code finishes. This allows us to prevent arbitrarily deep stacks and the potential for complicated, hard-to-debug race conditions.

Cross-capsule method calls allow loose coupling because they hide the implementation details of the capsule we are calling, allowing us to change the implementation at a later time, or even introduce a new capsule between the caller and the intended callee. Without this functionality, capsules that wanted to interact in this way would need to be environmentally related, which presumes knowledge of the inner operation of a capsule, or spurious extra state required only for the operation of the calls. Also, transparent insertion of proxy-objects and the like would become impossible.

For an example, let's assume we have a capsule *Prime-Finder* which is responsible for finding as many prime numbers as it can. We will also have a capsule *Prime-Tester* that actually does the grunt-work of testing a given number for primality. By using cross-capsule method calls to control the interaction of the two capsules, we can introduce new functionality or features later on with minimal effort. *Prime-Finder* calls *Prime-Tester*'s *test-prime* method with a number to test as the argument to initiate prime-testing. When *Prime-Tester* completes its execution, it calls *is-prime* with the number it tested and a boolean indicated whether the number is a prime. In the first release of the code, *Prime-tester* would do the computations itself. This mode of interaction would allow us to transparently implement an RPC layer that could allow the capsules to exist on different nodes. Likewise, we could change the implementation so that it farms out subspaces of the testing space to multiple other nodes. The point is that this provides us with abstraction which will allow us to build complex functionality through multiple layers of abstraction, the same as any other useful language.

2.3.8 Example: Getting Capsules to Interact

Now that we have a shared-memory mechanism, we can implement our gradient descent packet, which needed to be able to look into the gradient packet to make sure that it is going in the right direction.

Gradient Descent

The computing node attached to the object to be tracked periodically broadcasts a capsule. All the computing nodes that hear this capsule want to let the base-station know, so that it can try and approximate the location of the object we're tracking. There are three possible ways that we could do this. Each computing node could send a message to the base-station, resulting in a potentially non-trivial amount of network traffic. Alternatively, we could have all of the nodes that heard the broadcast work together to build up a list of who heard the broadcast, and send that to the base-station. This approach reduces network traffic, but may introduce additional delays between when the computing nodes hear the initial target broadcast and when the base-station finds out. The third alternative combines both prior approaches; every node that hears the object's broadcast sends a message to the base-station. The messages pool their knowledge of who heard the broadcast if two end up on the same node en route. This method results in slightly more network traffic than the initial consensus approach, but is more resilient to complicated network topologies caused by barriers throughout the space, as well as allowing the base-station to receive the message data more quickly. Although not all the data might arrive at the same time, this allows the base-station to perform iterative improvements as it receives additional data.

The following code introduces the desired gradient descent functionality.

```
(capsule gradient-descent (gradient)
   ((last-grad-val 255)
    (nodes-who-heard (list))
    (initial-broadcast #t))

   (init ()
     (if initial-broadcast
       ;; We're just hearing this from the tracked object.
       (seq
         (set initial-broadcast #f)
         (set nodes-who-heard (list (node-id)))
         (schedule-me (random-range SEND_DELAY_LOW SEND_DELAY_HIGH)))
       ;; We're en-route to the base-station...
       (if (= grad-val 0)
         ;; We've arrived. Let the base station know who heard the bcast.
         (seq) ;; (we'll fill this in later)
         ;; Otherwise, only rebroadcast if we're moving towards the base.
         (when (< grad-val last-grad-val)
           (schedule-me (random-range SEND_DELAY_LOW
                                      SEND_DELAY_HIGH))))))

   (tick ()
     (broadcast-me))

   (collide (resident incoming)
     ;; Only re-broadcast if the incoming packet knows about someone
     ;;  we didn't already know about. (We actually schedule, so if
     ;;  we were already scheduled, we wait a bit more, which helps
     ;;  any additional data we don't know about catch up.)
     (def all-nodes-who-heard (union resident:nodes-who-heard
                                     other:nodes-who-heard))
     (def new-nodes (diff all-nodes-who-heard
                          resident:nodes-who-heard))
     (unless (empty? new-nodes)
       (set resident:nodes-who-heard all-nodes-who-heard)
       (schedule-me (random-range SEND_DELAY_LOW SEND_DELAY_HIGH))
       (when (= grad-val 0)
         ;; give the base-station the updated roster.
        ))
     resident) ;; keep the resident capsule, regardless
   (packup ()
     (set last-grad-val grad-val)))
```

By defining *gradient* as its environmental parent, *gradient-descent* is able to access the *grad-val* stored in the *gradient* capsule on the current node. This allows *gradient-descent* to ensure that it is moving towards the base-station by keeping track of the last *grad-val* it saw. Also, it provides a guarantee that our capsule doesn't go wild and saturate the whole network.

The *init* method handles the two ways in which this capsule could come to a node for the first time. In the first case, the capsule was just broadcast by the tracked node, and is effectively saying "I'm here!" It is its responsibility to notify the base-station of this, so it can repurposes the capsule for the second case. The second case is that it is a capsule with a list of nodes who heard the tracked node making our way towards the base-station. The capsule always rebroadcasts itself in the first case, because otherwise the base-station would never find out that the current node heard the tracked node. The second case only rebroadcasts if the current node is closer to the base-station than the last node.

The *collide* method handles the situation when an instance of *gradient-descent* already exists on the node and another instance arrives. In this case, the code determines whether the incoming *gradient-descent* is telling it anything new. If it is, the code updates the resident capsule's list of nodes that heard the tracked node, and schedules the capsule to be rebroadcast. If no new nodes are mentioned, the code does nothing, suppressing unneeded broadcasts.

What we still need

All of our capsules have the same major failing, there can only be one of each capsule on a node at a time. This means that we can only ever send one command, emit one gradient, and send one gradient-descending notification to the base-station. As previously discussed, this limit exists because otherwise capsules might unintentionally multiply without bound, which is not a good thing. Although we could copy and paste each of these capsules as many times as we need, giving them a new name each time, that's clearly not an elegant or scalable solution. The solution is the previously mentioned "naming variables" which we'll now discuss in greater depth.

2.3.9 Naming Variables

Naming variables are variables that supplement the unique id of a capsule to determine its uniqueness. Without them, it would be impossible to perform anything more than trivial actions with any reliability. For example, let's say we had a capsule that tests the primality of a number. This means that one of the data fields would be the number we're testing. Let's assume we have a capsule that's testing the primality of 73, and another that is testing the primality of 75. Although these capsules are clearly not equivalent, without the existence of naming variables, RGLL would see them as equivalent. This means that only one of the capsules could exist on a node at a given time. If the 73-tester was on a capsule, and then the 75-test arrived, only one of the capsules would survive. Our *collide* method could do nothing to reconcile the two primality tests; they are clearly different entities and should be treated as such. This is what naming variables allow.

It is also important to point out what naming variables should not be used for. The best way to think of naming variables is like parameters to a function. In the primality-testing case, the parameter is the number we're testing. Internal variables to the capsule would include the last number we tried to divide our number by, and the result of the primality testing. Not so coincidentally, this state can be reconciled in case of a collision. By comparing the internal states of two primality testing capsules for the same number, we can decide who wins (the tester who has concluded the number is/is not a prime, or has tested the most potential divisors.) Of course, not all function parameters are essential for uniqueness, and using them as such may unnecessarily reduce efficiency. An additional way to think of naming variables is like unique keys in a database. This also makes sense, as capsules are in many ways similar to records. A database needs a way to disambiguate rows in a table, hence the need for unique keys. But this also saves us from having to use the whole record as the key; instead of talking about “John Jacob Jingleheimer Schmidt”, we just refer to the unique id. The same holds with naming variables.

Capsule-Level Special Forms

Now that we have naming variables, a few additional special forms are required to be able to create and manipulate capsules with naming variables.

specialize *capsule-name ((naming-var-name1 val) (naming-var-name2 val))*
Specialize is used on capsules who contain naming variables. The function creates a new instance of the given capsule, with the naming variables initialized as provided. All naming variables of a capsule must be provided. Non-naming variables may be optionally set if desired. The new capsule is placed in the in-queue of the local computing node. Some time after the completion of this function and the execution of the current code block, the in-queue will be processed, and the *init* method of the capsule executed, potentially after *collide* is executed if an equivalent capsule already exists on the current node.

specialized-ref *capsule-name ((naming-var-name1 val) (naming-var-name2 val))*
Returns a unique integer id referencing a specialized capsule with the given values if it exists. Returns #f otherwise. This provides the mechanisms required to be able to deal with capsules containing naming variables when they are not in your ancestral hierarchy. Because we cannot number all potential specialized capsules, nor would we seek to, the returned integer id is actually a sentinel value that is guaranteed to be a valid reference to the capsule until the next call to *specialized-ref.* These semantics allow us to consistently refer to capsules by integer ids, while also dealing with specialized capsules.

Environmental Effects

The existence of naming variables raises the issue of the existence of capsules whose parents define naming variables, and the potential complexities that ensue. For maximum flexibility, we allow the “children” of capsules with naming variables to contain any subset of their parent's naming variables, which includes none of them. If the child capsule defines all of the naming variables of its parent, then the relationship is one-to-one, and the child treats the parent like neither of them had naming variables. This follows intuitively from our definition of capsule inheritance. The tricky case is the one where the child only defines a subset of the naming variables of its parent. In these cases, a set is built up by taking all of the parent capsules whose named values match the named values defined by the child capsule. The inherited variables from the parent are presented as a list. If the child capsule uses the optional capsule:variable syntax to refer to its parents, then the capsule name becomes a list of the set of parents, with each item supporting nested reference, rather than having each of the variable names being a list of the values of the given variable in the set of parents.

The same logic holds true for the children (generation 3) of the children (generation 2) mentioned here, and continues for an unlimited number of generations, but a few clarifications are in order. The n^{th} generation's set of naming variables must be a subset of their immediate parents, generation *n - 1*. A potential result of this is that the sets provided by different generations will not be the same size.

The requirement that a capsule's ancestors must be present for the capsule to be executed are not removed in this case. In other words, if the set of parents is empty, the capsule will still not execute on an empty set of list values.

2.3.10 Example: Multiple Instances of Capsules

Now we use naming variables to allow us to issue multiple commands, multiple gradients, and multiple tracking requests. Besides having to introduce a naming variable to differentiate the different commands/gradients/requests, we also have to have a way to initiate them. We'll take care of both of these tasks now.

Multiple Floods

The following code provides an updated *flood* capsule to support multiple instances via a naming variable.

```
(defconst REBROADCAST_LOW  2)
(defconst REBROADCAST_HIGH 5)

(capsule flood ()
   ((flood-id    0 #t) ;; Naming!
    (target-node 0))

   (init ()
     (schedule-me (random-range REBROADCAST_LOW
                                REBROADCAST_HIGH))
     (when (= target-node (node-id))
       ;; Initiate the gradient.
       (specialize gradient ((grad-id (node-id))))))

   (tick ()
     (broadcast-me))

   (collide (resident incoming)
     ;; If we're already present, favor resident to stop the flood.
     resident))
```

As you can see, the changes to the capsule were trivial; all we had to do was add the *flood-id* naming variable. Although we could have used *target-node* as the naming variable in this case because we will only ever send a single node a single command, the use of flood-id is more generic, and scales to the case where we may send a node more than one command. Note that we don't actually use *flood-id*, it merely serves to differentiate our flood from other, logically distinct, floods.

We still need a capsule that can initiate the flooded commands from the base-station. Also, we haven't yet shown how the flood-id gets set. The following capsule solves both problems.

```
(capsule flood-commander ()
   ((next-flood-id 0))

   (do-flood (target-node)
    (specialize flood ((flood-id  next-flood-id)
                       (target-id target-node)))
    (set next-flood-id (+ 1 next-flood-id))))
```

Multiple Gradients

The following code provides an updated *gradient* capsule that supports multiple instances via a naming variable.

```
(defconst SEND_DELAY_LOW 2)
(defconst SEND_DELAY_HIGH 5)

(capsule gradient ()
   ((grad-id  0 #t) ;; Naming
    (grad-val -1)) ;; Deal with initial self-send on specialization

   (init ()
     (schedule-me (random-range SEND_DELAY_LOW SEND_DELAY_HIGH)))

   (tick ()
     (broadcast-me))

   (collide (resident incoming)
     (if (< incoming:grad-val resident:grad-val)
         incoming
         resident))

   (packup ()
     (set grad-val (+ grad-val 1))))
```

We already modified the flooded command to specialize and thereby initialize our gradient, so with the addition of the naming variable, we're all set.

Multiple Gradient Descent Packets

The following code provides an updated version of the *gradient-descent* capsule that provides functionality to support multiple instances.

```
(capsule gradient-descent (gradient)
   ((grad-id 0 #t) ;; Specialize down to one parent.
    (pulse-id 0 #t) ;; Naming!
    (last-grad-val 255)
    (nodes-who-heard (list))
    (initial-broadcast #t)
    (local-insertion #t))

   (init ()
     ;; The specialization means we'll be received by the tracker first,
     ;;  so ignore that.
     (if local-insertion
       (set local-insertion #f)
       (if initial-broadcast
         ;; We're just hearing this from the tracked object.
         (seq
           (set initial-broadcast #f)
           (set nodes-who-heard (list (node-id)))
           (schedule-me (random-range SEND_DELAY_LOW SEND_DELAY_HIGH)))
         ;; We're en-route to the base-station...
         (if (= grad-val 0)
           ;; We've arrived. Let the base station know who heard the bcast.
           (seq) ;; (we'll fill this in later)
           ;; Otherwise, only rebroadcast if we're moving towards the base.
           (when (< grad-val last-grad-val)
             (schedule-me (random-range SEND_DELAY_LOW
                                        SEND_DELAY_HIGH)))))))

   (tick ()
     (broadcast-me))

   (collide (resident incoming)
     ;; Only re-broadcast if the incoming packet knows about someone
     ;;  we didn't already know about. (We actually schedule, so if
     ;;  we were already scheduled, we wait a bit more, which helps
     ;;  any additional data we don't know about catch up.)
     (def all-nodes-who-heard (union resident:nodes-who-heard
                                     other:nodes-who-heard))
     (def new-nodes (diff all-nodes-who-heard
                          resident:nodes-who-heard))
     (unless (empty? new-nodes)
       (set resident:nodes-who-heard all-nodes-who-heard)
       (schedule-me (random-range SEND_DELAY_LOW SEND_DELAY_HIGH))
       (when (= grad-val 0)
         ;; give the base-station the updated roster.
        ))
     resident) ;; keep the resident capsule, regardless

   (packup ()
     (set last-grad-val grad-val)))
```

Now we also need another capsule that resides on the tracking node to initiate these pulses. The following code achieves this goal.

```
(defconst PULSE_INTERVAL 50)
(defconst BASE_STATION_ID 0)
```

```
(capsule tracker-pulser ()
   ((next-pulse-id 0))

   (init ()
     (schedule-me PULSE_INTERVAL))

   (tick ()
     (specialize gradient-descent ((grad-id BASE_STATION_ID)
                                   (pulse-id next-pulse-id)))
     (set next-pulse-id (+ 1 next-pulse-id))))
```

2.4 Functors

Without a method for providing capsule prototypes and higher-order manipulation of capsules, writing large-scale programs would be an exercise in futility, and would make the goals of RSEAM impossible. To this end, RGLL provides a functor implementation designed to allow things as simple as the specialization of a prototype capsule, to things as full blown as injecting code and variables into a capsule, or even generating a whole family of capsules around a single code block. If any of the following looks or sounds like a macro implementation, that's because it essentially is. However, unlike CLOS, RGLL itself is not capable of handling the rigors of a macro system. The result is that semantically we need to fall back on something equivalent, but different, namely functors.

2.4.1 Functor Definition

Functors are defined using the following syntax.

```
(functor FUNCTOR-NAME
 ;; Argument list.
 ((ARGUMENT-NAME1 ARGUMENT-TYPE)
  (ARGUMENT-NAME2 ARGUMENT-TYPE)
  ...)
 ;; Output list.
 ((OUTPUT-NAME1 OUTPUT-TYPE)
  (OUTPUT-NAME2 OUTPUT-TYPE)
  ...)

 (INIT-CODE...)
```

```
  TOP-LEVEL-DEFS...)
```

Top level definitions go in the body of the functor labelled “TOP-LEVEL-DEFS”. The top-level definitions are quoted verbatim, spliced into the list of top-level definitions where the functor is instantiated. The one exception is that anything preceded with an “@” is evaluated at instantiation time and replaced with the result of its evaluation. Additionally, before the top-level definitions are quoted and evaluated, the following things happen. First, all of the arguments are defined at instantiation time with the values provided at the instantiation. Second, the output variables are also defined at the same scope. Then the “init-code” is executed in the same instantiation-time scope. Finally, the top-level definitions can be copied and spliced into the top level, with “@” expressions evaluation and replacement occurring as the tree is copied.

The following types are applicable for arguments and outputs:

- <**int**> : Integer value.
- <**log**> : Boolean (logic) value.
- <**list**> : A list.
- <**block**> : A block of code. This can be something as simple as *1*, or as complicated as *(seq (def a 1) (set a 2))*. This should generally be considered for use on arguments only, as only functors know how to deal with blocks. (Although the value could be passed to another functor as an input.)
- <**capsule-name**> : The name of a capsule; in other words, the unique symbol that identifies a capsule. This can refer to either an existing capsule, or the name of a capsule that you would like created.
- <**capsule-def**> : Only applicable to inputs. Although provided as a symbol-name, just like <**capsule-name**>, the semantics of the variable inside the functor are changed to represent the contents of an existing capsule definition.

The variable can then be used with qualifiers as a nested reference. Assuming the variable was named *capsule*, *capsule:name* refers to the capsule's name, *capsule:env* refers to the environment definition, *capsule:parents* refers to the list of parents, *capsule:met-name* refers to the list of expressions comprising a given method on the capsule, and *capsule:mets* refers to the list of statements declaring all of the methods and their code as a whole.

Functor definitions are top-level constructs; they cannot be created inside capsules.

2.4.2 Functor Invocation

Functors are instantiated using the following syntax.

```
(functorize SAVE-NAME FUNCTOR-NAME
   ;; Populated arguments
   ((ARGUMENT-NAME1 ARGUMENT-VALUE)
    (ARGUMENT-NAME2 ARGUMENT-VALUE)
    ...))
```

The "save-name" is a symbol name that should store the output results of the functor. When used with nested reference qualifiers, the individual outputs of the functor are available. If a functor has outputs *foo* and *bar*, and we instantiate the functor with a save-name of *abba*, then we can reference the outputs as *abba:foo* and *abba:bar* throughout the rest of our code. They will be treated as read-only constants which resolve at compile-time.

Arguments are specified with their values in a key/value syntax. It is quite possible that there will be several arguments, each of which may take up several lines. Without the inclusion of argument names it would be significantly harder to read and understand the functor invocations, as well as maintain them.

2.4.3 Helper Functions and @ Syntax

Some tasks, such as generating new unique capsule-names, require that we have some way of executing code when functors are invoked by functorize. Much like Goo's "unquote"/"," when used in the context of quasiquote, the @ actually indicates that the expression that follows should be executed when the functor is invoked, and its result should replace the expression before the body of the functor is parsed. In other words, if we have the statement (set foo @bar) in a functor body, and *bar* is an argument with value 1, then this will be converted to (set foo 1). We can also do something like (set foo @(+ bar 1)), which will be converted to (set foo 2). When the result of evaluating the expression is not a list, the value is placed in the spot the expression occupied. However, when the result is a list, the contents of the list are spliced in, not nested. For example, if *abc* is an argument with the value *(list 1 2 3)*, then the expression *(+ @abc)* when located inside the top-level definition segment of the functor would expand to *(+ 1 2 3)*, not *(+ (1 2 3))* or *(+ (list 1 2 3))*.

The following additional functions are also provided to aid in the creation of useful functors. They are only available at functor invocation time, and are not available to code capsules.

catsym *symbol1 symbol2 ...*

Concatenates the given list of symbols and returns a new symbol. Useful for generating unique or derived capsule names.

gensym

Generates and returns a unique symbol. When generating unique symbol names for use in derived capsules, you may want to concatenate the generated symbol onto an existing symbol with meaning to aid in debugging.

Also, all of the RGLL special forms that do not involve capsules are available. In particular, this means that arithmetic and list operations can be performed.

2.4.4 Example: Reuse via Functor-Provided Prototypes

There are still a few things we need to take care of before our example really works. The first problem we need to deal with is how to get the gradient capsules to the anchor nodes in the first place. Although the flooded commands are "self-propelled", they depend on a gradient capsule already existing on the anchor node that can be specialized. One valid solution to this problem would be to have the base-station emit a gradient before it notifies any anchors of their status. The problem is that this is basically a hack, and is not a viable solution in most other cases. For this reason, we will introduce a "codecast" capsule that ferries code capsules throughout the network. Since this behavior is apparently something that we might need to implement frequently, it makes sense to also provide a simple, reusable way to introduce this functionality into our code. The solution is, of course, functors.

Codecast

Codecast has a list of capsules that it is responsible for distributing throughout the network. It has two active modes of operation, sending and receiving. In the sending mode it maintains a list of capsules it needs to send, while in receiving mode it keeps track of the capsules it has yet to receive. Because our capsules are intended to be deployed in an environment where broadcast is not reliable, it is not sufficient for *codecast* to simply send each capsule once. For that reason, capsules in the receiving mode periodically broadcast themselves so that the sender can update its list of capsules that still need to be sent. The sender actually only transmits itself once it has transmitted all of the payload capsules, but the receivers broadcasting themselves to update the sender's state can also act as a propagation mechanism as well as a hedge against lost *codecast* capsules.

The following code implements codecast.

```
(defconst STATE_INACTIVE 0)
(defconst STATE_SENDER   1)
(defconst STATE_RECEIVER 2)

(defconst SEND_TICK_MIN 3)
(defconst SEND_TICK_MAX 5)

(defconst RECV_TICK_MIN 10)
(defconst RECV_TICK_MAX 15)

(functor codecast
   ((roster <list>))
   ((codecast-name <capsule-name>))

   @(set codecast-name (catsym 'codecast- (gensym)))
```

```
(capsule @codecast-name ()
   ;; State
   ((payload @roster)
    (checklist ()) ;; capsules present (receiver)/to send (sender)
    (state STATE_SENDER))

   (init ()
     ;; Receiver:
     (when (= state STATE_RECEIVER)
       ;; Take Inventory
       (call inventory)
       ;; Start tick to indicate our state a ways in the future
       (schedule-me (random-range RECV_TICK_MIN RECV_TICK_MAX))))

   (inventory ()
     ;; Inventory
     (for ((capsule payload) (present checklist))
       (set present (capsule-present? capsule)))

     ;; See if we should transition to SENDER...
     (unless (find ((present checklist))
               (not present))
       (set state STATE_SENDER)))

   (tick ()
     ;; Sender: (more frequently)
     (when (= state STATE_SENDER)
       ;; Find someone we haven't sent yet, and send them.
       (def nextcapsule (find ((tosend checklist) (capsule payload))
                          (when tosend
                            (set tosend #f)
                            capsule)))
       (if nextcapsule
          (broadcast nextcapsule)
          (seq
            (broadcast-me)
            (set state STATE_INACTIVE)))

       (schedule-me (random-range SEND_TICK_MIN SEND_TICK_MAX)))

    ;; Receiver: (less frequently)
    (when (= state STATE_RECEIVER)
      ;; Take inventory some more.
      (call inventory))

    ;; (may no longer be receiver...)
```

```
     (when (= state STATE_RECEIVER)
       ;; Broadcast ourselves to provide an update of what we still need.
       (broadcast-me)

       (schedule-me (random-range RECV_TICK_MIN RECV_TICK_MAX))))

    (collide (resident incoming)
      ;; Sender:
      (when (and (or (= resident:state STATE_SENDER)
                     (= resident:state STATE_INACTIVE))
                 (= incoming:state STATE_RECEIVER))
        ;; Use this to update the state of things we need to send.
        (for ((tosend resident:checklist) (received incoming:checklist))
          (set tosend (or tosend (not received))))
        (set resident:state STATE_SENDER)
        (schedule-me (random-range SEND_TICK_MIN SEND_TICK_MAX)))

      ;; Receiver:
      ;; We can safely ignore new instances of ourself.

      ;; We always keep ourselves...
      resident)

    (packup ()
      (when (= state STATE_SENDER)
        (set state STATE_RECEIVER)))))
```

We still need to instantiate this functor, creating a specialized codecast capsule designed to spread our *gradient* capsule throughout the network. The following code snippet implements this.

```
(functorize gradientcast codecast
   ((roster (list 'gradient)))) ;; Input
```

A capsule can use *gradientcast:codecast-name* if it needs to refer to the generated capsule.

2.4.5 Example: Reuse via Functor-Provided Mix-Ins

We also need to deal with the problem of capsules that are no longer needed clogging up our computing nodes. A side effect of using naming variables is that unless the space that the naming variables spans is very small (i.e. the naming variables are booleans), there is no way our computing nodes could store all the possible capsules. Most of the time, we don't need our specialized capsules for an extended period of time; they accomplish their required task, and then we can purge them from the system. However, it is important to keep in mind that we may need to keep capsules around long enough to ensure that they die out.

Let us take flooded commands, for example. Strictly speaking, the use of the flood is done once it reaches the intended target. However, only the target knows when it has been reached. Having the target send out a kill-variant of the capsule to purge is not a good solution because it has non-trivial message overhead, will never happen if the target does not exist or cannot be reached, and most importantly, could very easily cause a ghost-ship effect, as the killer capsules ricochet throughout the network forever. A better solution is to have the capsules delete themselves after a period of inactivity. The length of the period is lower-bounded by the diameter of the network. For example, in a network topologically shaped like a torus, we can end up with the equivalent of a dog chasing its tail. In a pathological case, we end up with the broadcast wavefront moving around the torus, chasing the wavefront of capsules terminating themselves. Such a phenomenon would continue indefinitely.

Timed Purge

Our timed-purge functor takes a capsule as input, and redefines the capsule. The redefined capsule has a new method, *purge*, that is scheduled to be executed when the capsule arrives, and rescheduled any time a tick or collision occurs. Remember that if a capsule already is scheduled for execution, and a new scheduling request is made, the original request is invalidated and then the new scheduling request is made.

The intent is to be able to provide an automatic purging mechanism to capsules without having to manually add the functionality to every capsule that needs it. All that is required is that we invoke the functor on a capsule after the capsule has been defined. By injecting our scheduling code into the main entry points to the capsule, we can track activity, and delay the purge until the capsule has not seen any activity for a given number of cycles. Although the logic for *init* and *tick* is pretty clear, the rationale for injecting our code into collide may not be quite so obvious. If a collision occurs, either the existing capsule will do work, implicitly act as a suppression mechanism, or allow the incoming capsule to overwrite it. In the first two cases, there is clearly activity afoot that indicates we should delay the purge, and in the third case, it doesn't matter what we do.

The following code defines a functor that implements the desired timed purging behavior in capsules that are passed in.

```
(defconst PURGE_DELAY 100)

(functor timed-purge
   ((our-cap <capsule-def>))
   ()

   ;; Replace the original capsule.
   (capsule @our-cap:name (@our-cap:parents)
      (@our-cap:env)

      (init ()
        (schedule 'purge PURGE_DELAY)
        @our-cap:init)

      (tick ()
        (schedule 'purge PURGE_DELAY)
        @our-cap:tick)

      (collide (resident other)
        (schedule 'purge PURGE_DELAY)
        incoming ;; default behavior in case collide was not defined.
        @our-cap:collide)

      (purge ()
        (purge-me))

      @our-cap:mets))
```

We can introduce this functionality into our flood capsule as follows.

```
(functorize garbage timed-purge ((our-cap 'flood)))
```

We use *garbage* as the save name because it has no outputs, and even if it did, we wouldn't care what they are in this case. There is no magic to choosing *garbage*, it merely conveys the unimportance of the result.

2.4.6 Functor-func Syntax

As much fun as using functors is, the *functorize* syntax can still be a bit inconvenient in many situations. Functor-func's are a form of syntactic sugar with a few additional features. Functor-funcs allow you to instantiate a functor inside a capsule like it was a normal function, and have automatic "glue code" inserted by the functor-func to link the current capsule with any capsules generated by the functor-func. Functor-funcs provide functionality that is similar in spirit to Java's anonymous inner-classes. Java's anonymous inner-classes let you create a new class from within your current class without having to go through the hassle of constructing a new file to hold a new class that does something trivial. (Ignore the fact that Java inner-classes also have the ability to access the private and protected members of the class in which they are declared.) The result is that although functor-func's don't let us actually do anything we couldn't do before, they let us do it with less typing, and potentially more encapsulation, as the specific details of interacting with a functor-generated capsule can be hidden away.

The *functor-func* syntax is as follows.

```
(functor-func FUNCTOR-FUNC-NAME
   ;; Argument list.
   ((ARGUMENT-NAME1 ARGUMENT-TYPE)
    (ARGUMENT-NAME2 ARGUMENT-TYPE)
    ...)

   (INIT-CODE...)

   ;; Environment additions to invocation capsule
   ((NEW-VAR-NAME1 NEW-VAR-DEFAULT-VALUE1)
    ...)

   ;; Additional methods to define
   ((METHOD-NAME1 (ARGS) ...)
    ...)
```

```
  ;; Code to place at invocation point
  (INVOCATION-POINT-CODE...)

  TOP-LEVEL-DEFS...)
```

As you can see, the syntax is somewhat similar to the standard functor syntax. We remove the list of output variables, and add three new mandatory items. The first is a list of variables that should be added to the environment of the capsule in which the functor-func is invoked. The second is a list of methods to be added to the capsule. The third is a sequence of code to be injected at the invocation site.

2.4.7 Functor-func Invocation Syntax

The syntax for *functor-func* invocation is just your standard, run-of-the-mill function invocation. If we have a functor-func *foo* with arguments *bar* and *baz*, defined in that order and both of which are integers, then we can invoke the functor-func like so: *(foo 2 3)*. This invocation must occur inside the definition of a method inside a capsule definition. For a more detailed example, see the next running example section.

The syntax disparity between invocation of a functor-func, where the functor-func name is the de facto function of the expression, and the invocation of a function on a capsule, where *call* or *call-other* must be used, is intentional. The use of *call* on same-capsule functions is actually debatable, because the semantics of the call are traditional call semantics; the currently executing code is suspended while the other function is invoked, and upon completion of that code, control is returned as well as the return value of the code. *Call-other*, however, has decidedly different semantics, acting more like a message-passing construct. What is key is these functions are not primarily intended as a means of abstraction; these are implementation details, and a syntax that tries to conceal the reality of the situation is not helpful. Functor-func's semantics, however, are intended as a means of abstraction, allowing us to build higher-level functionality without getting lost in a morass of implementation details.

2.4.8 Functor-func Invocation Semantics

Although the sequence of events that occurs when a functor is invoked/instantiated are relatively clear, the functor-func case is slightly more complicated because it is invoked in the middle of a capsule, not at top-level. However, the semantics are actually roughly equivalent to the functorize case; the top-level definitions are actually evaluated at the time of functor-func invocation. For most purposes, exactly when the top-level definitions are evaluated is unimportant. The capsules defined by the functor-func are not made directly visible to any of the code in the functor-func invocation's enclosing capsule. The variable and method additions to the capsule have no real sequence, as method definitions and variable definitions occur in parallel to each other from a semantic level. The injected code is, by definition, injected at the invocation site. Constant definitions are the exception to the rule. A constant defined in the functor-func will not be available to the code executed prior to the functor-func's invocation, but will be available to the code after its invocation.

One question that is raised is how RGLL knows to expand functor-funcs since the invocations are not top-level definitions. The answer is that functor-funcs essentially define special-forms. RGLL compilation moves through three phases. First, a Lisp-style reader parses the program code into nested lists. Then, RGLL consumes the lists, transforming them into an AST. This AST is what is later transformed into binary code. The production of the AST relies on special-forms consuming lists and producing AST nodes. This is when functor-func's and functors alike are instantiated. The functor-func invocation is evaluated, returning the AST sub-tree corresponding to its injected code, splicing in evaluated top-level definitions in the functor-func to the top-level AST, and mutating in-memory structures as appropriate.

2.4.9 Example: More Convenient Reuse through Functor-func's

At this point, our tracking demonstration is fully operational. However, it is not as cleanly implemented as it could be. We claim that RGLL is scalable and provides abstraction mechanisms, yet our flooded command is completely hand-rolled. In this problem-domain, it seems like flooded commands might be a common occurrence. This is something that's just begging for reuse. Although we could use a standard functor to do this, a functor-func is much more convenient to use, and significantly more expressive.

The following code defines a functor-func for creating targeted flood commands.

```
(functor-func flood-targeted-command
   ((target-node-var <block>)
    (payload <block>))

   ;; Init code (variables live for the duration of instantiation)
   ((def capsule-name (catsym 'flood- (gensym))))

   ;; Environment additions
   ((next-flood-id 0))

   ;; Additional Functions
   ()

   ;; Invocation Point Code Injection
   ((specialize @capsule-name ((flood-id next-flood-id)
                               (target-node @target-node-var)))
    (set next-flood-id (+ 1 next-flood-id)))

   ;; Top-level defs...
   (defconst REBROADCAST_LOW  3)
   (defconst REBROADCAST_HIGH 5)

   (capsule @capsule-name ()
    ((flood-id    0 #t) ;; Naming!
     (target-node 0))

    (init ()
      (schedule-me (random-range REBROADCAST_LOW
                                 REBROADCAST_HIGH))
      (when (= target-node (node-id))
        @payload))

    (tick ()
      (broadcast-me))

    (collide (resident incoming)
      ;; If we're already present, favor resident to stop the flood.
      resident)))
```

Our *flood-targeted-command* functor-func takes 2 arguments. The first argument is a block of code that should evaluate to the node id of the intended target of the flooded command. The second argument is the payload that should be executed on the node when the command reaches it. The init block of the functor-func is used to set up any variables that we will need to use "@" to reference. In this case we create a capsule name for the capsule we're going to generate so we can reference it later. Our functor-func will add one variable to the capsule environment in which it is invoked, so that it can keep track of the next unique flood id to use. The behavior is simple enough that no additional functions are required, however. Our invocation point code is responsible for specializing the flood capsule and updating the variable that tells us the next unique flood id to use. If you recall, this little snippet of code used to be part of our *flood-commander* capsule, but is now abstracted out. The top level definitions in the functor-func define the actual capsule that will be flooded throughout the network. It uses the "@" syntax to inject the name we previously made up and the payload code that was provided as an argument.

Now we look at our revised *flood-commander*, which is also the only code we need to write at all related to the targeted flood. The code follows.

```
(capsule flood-commander ()
   ()

   (do-flood (target-node)
     (flood-targeted-command target-node
       (specialize gradient ((grad-id (node-id))))))
```

The *flood-commander* capsule still provides a *do-flood* method, which takes the *target-node* as an argument. The rationale continues to be that the host will send a capsule over its communication link to our base-station node that will invoke this method with its desired target. However, instead of specializing a flood capsule whose internal details we must know about to be able to do so, we now just invoke *flood-targeted-command* which was generated by the functor-func. All we need to provide is a block of code that resolves to an integer id that matches the unique node id of some node in the network, and the block of code that should be executed if/when the flooded command finds that node. We need know nothing about the creation of any capsules or how those capsules work, just the contract between us and the functor-func.

The benefit of this abstraction is even more clear when we compare it with an implementation that uses a functor instead of a functor-func. There are three ways we could achieve the same functionality with a functor.

- Have the functor construct the *flood* capsule, and return its name via an output variable. This burdens the *flood-commander* with the task of specializing the flood capsule, requiring it to know the variables internally used by the *flood* capsule. We must also instantiate the functor prior to defining *flood-commander*. The result is not a very clean solution.

- Have the functor construct the *flood* capsule and the *flood-commander* capsule. In order to provide any functionality in the *flood-commander* capsule beyond the flooding, additional functor arguments are required. This provides odd semantics to the functor, which would take in multiple blocks, and generate two capsules, each with a subset of the different blocks. Also, the functor would presumably impose the calling convention to be used by the base-station. This too is an awkward solution; the code will not be easy to read, and it unduly limits our implementation.

- Have the functor consume an existing *flood-commander* capsule, modifying it to invoke the *flood* capsule which the functor also creates. This is potentially more workable than the previous case, but has even more awkward semantics. In order to get the user's *flood-commander* code to inter-operate with the code we inject, we would presumably inject a new function equivalent to the *do-flood* method of our original implementation. This is awkward because the user would need to define a capsule that calls a method that does not yet exist. Then they would instantiate the functor which defines the *flood* capsule and redefines *flood-commander*. All in all, this would be a very odd solution.

Note that we could instantiate our timed-purge functor inside of the functor-func to add the timed-purge features to our flood capsule. There's no benefit to abstractions and the like if you can't stack them. The instantiation was omitted purely to keep the example to the point.

2.5 Scalability

The most important goal of RGLL and the inherent programming paradigm is that it must allow programs to easily scale. The problem of existing languages previously mentioned is that they tend toward hard scale-line boundaries. For TinyOS, the hard limit is defined by the resource limitations of the mote executing the program. For paintable computing, this hard limit is first reached when your program code reaches the maximum size of a process fragment or your data exceeds the available home page dimensions; however, because you can break a larger program into many overlay-style blocks and use the home page as shared memory with relative ease, there is a second, harder limit that is reached when your program occupies all available program fragment spaces on the computing nodes. In either paradigm, development is relatively easy until you hit the boundary. Before you hit the boundary, you can add new functionality with few concerns. However, once the boundary is reached, there is no clear and simple way to add the new functionality.

Remove Procedure Call (RPC) is the knee-jerk, quick-fix solution to such problems. Although it is quite a useful tool, RPC is not a panacea. First, RPC is not truly transparent, introducing new failure-modes to function calls where there were none previously. Secondly, RPC is only a realistic option on calls between loosely coupled modules with no global dependencies. A single module cannot be split down the middle and be expected to work via an RPC bridge without major modifications. Since the displaced functions are presumably coupled with the rest of the module via shared data, either protective getter's and setter's must be injected which use RPC to communicate with the original code, or both sides of the equation must be rewritten to pass all needed data back and forth. Clearly, both of these techniques are untenable unless you already have well designed, loosely coupled application-level modules. Application-level is the key term here, as you can't farm out your hardware abstractions, network code, or the RPC infrastructure that builds on them to another node.

Eventually, a point will be reached when a single module must be refactored into multiple smaller modules that can be bridged via RPC, or optimized for size. Neither solution is particularly attractive. Refactoring on such a scale is tantamount to a rewrite, and besides being a time-consuming task, could potentially introduce new bugs. Optimizing code to reduce its size could initially yield non-harmful benefits, but the more optimization performed, the greater the likelihood that clarity of implementation will be lost, setting the stage for new bugs to be introduced as the code is maintained.

You could easily and effectively argue that anyone who allowed their programs to grow to such a size without continually refactoring the code and modularizing it (a la extreme programming) as needed deserves what they get, and that this is not inherently a limitation of the TinyOS development paradigm. Clearly, a programmer with his wits always about him can amortize his effort over all his development and avoid these pitfalls. Although this may be true, most programmers follow the path of least resistance as provided by their language and libraries. It's best that the path of least resistance doesn't lead the programmer to shoot himself in the foot.

So how does RGLL help you avoid shooting yourself in the foot? A program written in RGLL is just as limited by the memory restrictions of its computing nodes as any other language. (In fact, RGLL is designed to be implemented on top of TinyOS.) RGLL does this in several ways:

- Constrains you to think in bite-sized pieces for both code and data. In the case of data, this encourages an object-oriented style of data storage, with tightly coupled data going together in the same capsule. This simplifies data-passing between computing nodes by encouraging it from the get-go. In the case of code, this makes it much simpler to distribute execution; a lengthy series of statements will be transformed into a series of capsules interacting through well-defined communication channels.

- Data dependency is made obvious by named parents, which can be used to help us know how to cluster processing code across nodes or allow us to move only the required data among nodes, without the use of fetch/store on demand across nodes.
- Provides mechanisms to add new, reusable functionality to existing implementations without modifying the original code definition.
- Allows abstraction layers to be easily stacked to create complex implementations.

2.6 Implementation

Virtual Machine

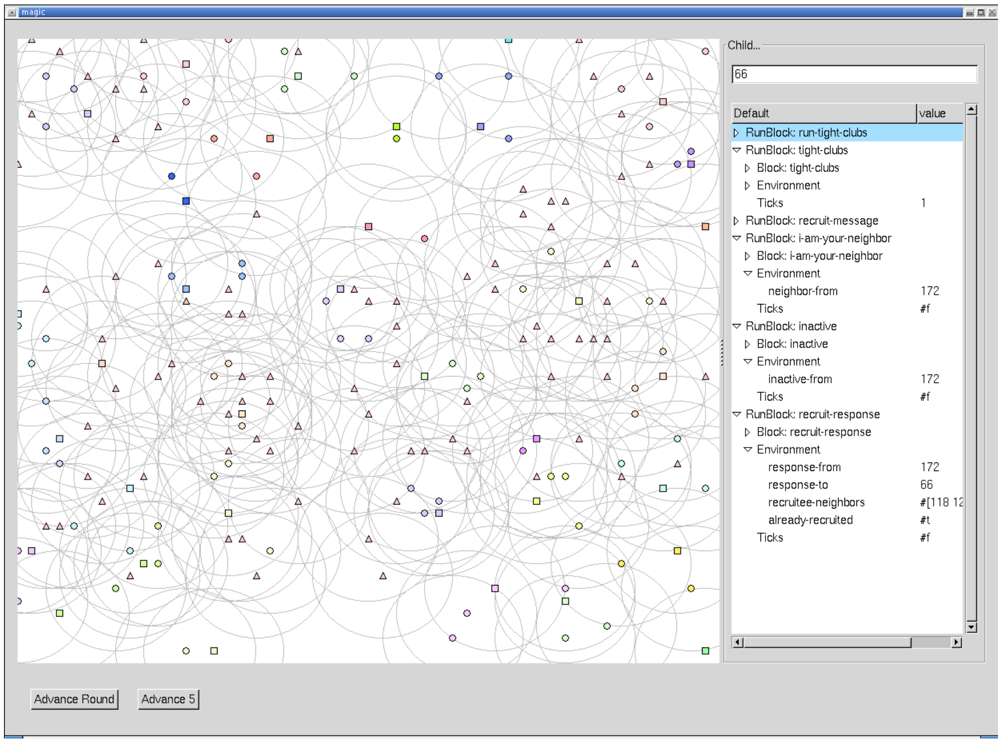
The virtual machine is implemented in Goo. An AST is built up from a tokenized list using Goo's reader.

Simulator

The simulator is implemented in Goo using the SamurUI GUI library.

The model used to represent the world is currently the typical ideal amorphous computing simulation environment. Nodes are placed in a 2-D coordinate system. Messages sent by motes are heard exactly by all motes within radius r.

Simulation is organized into rounds. From a timing perspective for the simulator, one tick is equivalent to one round. Each mote has a queue of incoming messages for each round, and a queue of outgoing messages. When a round is executed, the motes first process their incoming message queue, and then process their tick queue, executing the tick methods on any code capsules whose tick count is up. At the conclusion of the round, messages from the output queues are distributed using the simulation rules previously mentioned. Then the cycle begins anew.



66

```
Default                                value
▷ RunBlock: run-tight-clubs
▽ RunBlock: tight-clubs
   ▷ Block: tight-clubs
   ▷ Environment
     Ticks                             1
▷ RunBlock: recruit-message
▽ RunBlock: i-am-your-neighbor
   ▷ Block: i-am-your-neighbor
   ▽ Environment
        neighbor-from                  172
     Ticks                             #f
▽ RunBlock: inactive
   ▷ Block: inactive
   ▽ Environment
        inactive-from                  172
     Ticks                             #f
▽ RunBlock: recruit-response
   ▷ Block: recruit-response
   ▽ Environment
        response-from                  172
        response-to                    66
        recruitee-neighbors            #[118 12
        already-recruited              #t
     Ticks                             #f
```

Figure 2-2: An example simulator session of tight-clubs.

The user interface of the simulator displays the nodes in their given coordinate space. The communication radius of each node is superimposed on the image. The color and shape of each node is determined via a mapping process that is easily controlled using capsule-specific code logic. Simulation can be advanced using buttons. At any point, the user can click on a node in the display and be presented with a list of all the code capsules present on the node in a tree view. As shown in figure 2-2, each capsule entry can be expanded to show more information, included the current environment of the capsule, whether the capsule's *tick* method is scheduled (and when), the default environmental state of the capsule, the environmental parents of the capsule, etc.

Chapter 3

RSEAM

The goal of RSEAM is to allow serial programs to be compiled into a form that allows them to resiliently execute on an amorphous computing substrate. This chapter is a speculative discussion of how such functionality could be implemented. Sketches of implementations and description of what aspects of RGLL may be required are provided. Some of the components mentioned have already been implemented in RGLL, but by and large, everything that follows is hypothetical.

3.1 Concepts

3.1.1 Introduction

Two properties are required to guarantee that a program will run to completion:

1. Any work that is done should not be lost.

2. Execution should always be moving forward.

A corollary to the second property is that we do not want the book-keeping efforts of the system to swamp out the actual processing efforts. This is especially true in the cases of hardware failures. Because we expect the computing motes individually to be unreliable, we design our implementation to be minimally affected by random node failures. Larger-scale failures should also have minimal side-effects, but we accept that large-scale failures are non-trivial events, and as such, a fair amount of book-keeping overhead is expected with such events. However, such events should not prevent the successful execution of our program.

These properties provide guiding principles for RSEAM, and are expressed through several key concepts.

To these ends, we choose a method of grouping nodes into *cells*, which is the base semantic grouping of RSEAM. By conceptually working at a higher level than computing nodes, we provide a degree of resilience against individual computing node failures. However, to achieve this, we also need an implementation level granularity, and for this we use RGLL code capsules.

Even with some degree of abstraction from the actual hardware we operate on, it is still important to ensure that computation is always moving forward. Because no single node can hold the entire program, we expect that code capsules will frequently need to pass their results onto other capsules, or request calculations to be performed by other capsules and then use the returned values. However, a capsule cannot simply request that another capsule do something and then assume that it will be done successfully. The capsules exist in an inherently lossy medium; communications errors may result in their destruction in-flight, and computing node failures may result in their destruction when they think they are home and dry. RSEAM's solution to this problem is the *promise*, which provides a mechanism to ensure that a computation completes. A promise ensures that a called capsule/task runs to completion by restarting the task if it dies. Just as a standard program may have many calls in progress and therefore many frames on its stack, a running RSEAM program may have many outstanding promises at any given time.

However, promises merely form a resilient downward chain. If the first rung of the chain is destroyed, or, worse yet, the entire chain is destroyed via a large-scale deterministic node failure, we are sunk. For this reason, multiple *code repositories* are scattered throughout the network. The code repositories ensure that all of the first links of all of the chains are present, and if they are not, they restart them. Together with promises, the code repositories serve to ensure that execution is always moving forward.

Unfortunately, if we are constantly having to start from scratch, it doesn't help that execution keeps moving forward. *Checkpoints* store important calculations and state information so that we don't have to start from scratch every time. Completed calculations will store their results in the checkpoints. Likewise, long-running code will periodically save internal state.

Together, these conceptual mechanisms achieve the goals of RSEAM. More detailed descriptions follow.

Cells

Cells provide the base level of semantic granularity for RSEAM. Tasks are assigned to cells, not to individual nodes. We refer to the piece of a program that executes on a cell as a "logic cell." So when we refer to the code of a logic cell, we refer to all the capsules that make the operation of the given task on any logic cell possible.

Each cell is comprised of a group of computing nodes, each of whom can hear every transmission made by ever other member of the cell. This means that any node in the cell can be made the "leader". Because the leader is not topologically irreplaceable, if it were to fail, it can be replaced by any other operating node in the cell. If this were not the case, the failure of a leader in a cell could potentially require a reallocation of nodes in the network to groups, or leave otherwise usable nodes idle. Although power consumption is not a specific focus of the RSEAM implementation, this attribute potentially allows for power conservation (when the full resources of the system are not required) by having nodes within a cell rotate the leadership burden allocated to the cell.

Promises

The most important concept in RSEAM is that of the promise. A promise is a remote procedure call adapted to the lossy computing environment RSEAM is designed to execute in. Promises are responsible for locating the code that is to be executed, starting the execution, verifying the execution completes, and returning the answer, if any, to the caller of the promise. There are actually two levels of promises: one below the RSEAM level that allows for the failure of calls when the code to be executed can not be located, and one at the RSEAM level that guarantees success. The RSEAM level promise is built on top of the sub-RSEAM promise, and integrates with code repositories to re-introduce the required code into the network.

Code Repositories

Code repositories serve two important purposes. The first is that they contain the entirety of the capsules present in a given RSEAM program. This ensures that we never are left missing a piece of the puzzle. In a worst-case scenario where all nodes outside of a code repository have "amnesia", resetting them to their factory defaults, a single complete code repository should be able to restart the program completely from scratch. Note that a code repository may span multiple contiguous cells, depending on the size of the code base in question.

Secondly, code repositories ensure that all of the top-level execution capsules are executing. From a reachability perspective, code repositories are responsible for the static roots. The static roots are responsible for ensuring the execution of their own children, which makes sense because they are only reachable from the root itself. However, because there is no one looking out for the static roots from a reachability perspective, the code repositories must do so.

Checkpoints

Checkpoints exist to store successful calculations so that if node failures occur, we do not need to recalculate things we have already calculated. The question of the culling of checkpoint data is raised implicitly by the existence of checkpoints. After all, most non-trivial programs go through many more states than can be expressed by the output of the program. Even if it were possible to store the entire intermediate state in the checkpoints, there are no benefits to the programs from doing so. (Although it might be interesting from a debugging or instructional perspective.) As such, it is clear that useless information must be culled from the checkpoints to minimize resource requirements and increase efficiency.

This is accomplished by the RSEAM execution model. We can visualize the execution of transformed serial code as a tree of sorts. A linear run of code is converted into a horizontally chained group of capsules. As execution progresses, the active control moves from left-to-right across our chain of capsules. A function call makes a vertical call downward from a node in the horizontal execution group; the caller pauses execution, and defers active control to the new horizontal execution chain. When this execution chain runs to completion, it returns control to the original caller one-level up. This can be graphically imagined as a kind of call-tree. Once a horizontal execution-run completes, we have the result of the function, which can be checkpointed, and the original values can be purged. In the case of a long-running horizontal (sequential) executions, it is very likely that our code will have checkpointed. The new checkpointed value (associated with the caller), contains the logic required to remove the checkpointed data of the capsules that resulted in its creation.

3.2 Implementation

3.2.1 Cell Formation

AI Memo 1614 [CNW97] presents an algorithm dubbed *tight-clubs* that implements the grouping properties previously described as desirable in the concepts section. A quick sketch of their algorithm is presented here. For a full description of the algorithm, please consult the paper.

- First, each node gathers the unique ids of all its neighbor nodes.
- Each node maintains a set of all of its active neighbors, initially set to the set of its neighbors.
- If a node is not in a group and it has the lowest unique id of itself and all its active neighbors, the node becomes a leader, and it defines the potential recruits to its group to be the current set of all its active neighbors.

- Leader nodes attempt to recruit nodes in their set of potential recruits by sending them a message asking them to join. The picking heuristic chooses the node with the lowest unique-id of the potential recruits.

- Nodes not in a group respond to recruitment requests by joining the group and telling the leader its set of active neighbors. Additionally, they send a message to their neighbors indicating that they are now inactive.

- Nodes in a group that receive additional recruitment requests let the additional recruiters know that they are already in a group; the recruiter responds by removing the node from its set of potential nodes and going on to the potential recruit with the next lowest unique id.

- If a leader receives a successful recruitment response from a node, it intersects its set of potential recruits with the recruitee's set of active neighbors. This ensures that any future recruits can be heard by all the nodes currently in the group.

- If a leader's set of potential recruits is empty, the leader is done, and broadcasts a message indicating that it is inactive.

- If a node receives a message indicating that another node is inactive, it removes it from its list of active neighbors. The result is that given the previous rules, a node that is not part of a group may become a leader if a leader who was one of its neighbors completes its recruitment process.

.

At the end of execution, every node is a member of a tight-club, where every member can hear the broadcasts of every other member, assuming a constant broadcast radius and strength.

The algorithm can be augmented to deal with message dropouts in implementation by allowing the neighbor-meeting stage to take place for several cycles and having recruitment requests be re-issued if they time out.

3.2.2 Task Allocation

Equally important as forming cells is the allocation of tasks to those cells. A proposed method of task allocation is to use a force-directed graph model, where by we inject special capsules into the network that represent the execution cells that will be required. These capsules will represent the underlying communications links between cells as affinity for capsules, and otherwise will seek to repel themselves from other capsules. Using propagated gradients, the desired result is that the data-flow graph will arrange itself on cell substrate with unallocated cells throughout. These unallocated cells can then be used for code repositories and checkpoints; some will also be left unallocated, preferably around active cells. This provides future locations for code to be deployed in case of network damage, as well as insulating cells from the potentially collision-inducing radio chatter of their neighbors.

3.2.3 Routed Messages and Message Paths

Routed messages operate by establishing a trail of "bread crumb" capsules that mark the path from one capsule to another capsule. Once this path has been established, the nodes at either end of the path can send a message to the node at the other end of the path, and the cost of sending routed messages along the path is relatively low. However, establishing a message path is a relatively expensive proposition in terms of initial traffic. The initiator of the message path must emit a gradient in an attempt to locate the intended target. Once the gradient reaches the target, we may opt to wait until we feel the gradient has propagated sufficiently. When the distance weight assigned to a gradient is simply the hop-count, the delay should be minimal, just enough time to allow for the variability of propagation rates when using random delays. However, it is also possible that our gradient metric may be weighted using other factors, such as the busyness of the nodes the gradient is passing through. For example, if the most direct route via hop-count is through the middle of a busy processing cell that does not contain the target node, but there are unassigned nodes to either side of the cell, it would make more sense to route around the cell. This can be accomplished by having nodes in an assigned cell increment the distance metric by a greater amount. Additionally, it might make sense for the motes immediately surrounding the active cell, which are capable of hearing the bulk of the intra-cellular messages, to also assign themselves a greater distance metric. In this way, the message path can be steered away from computationally and communicationally active message paths.

When the target has been reached by the gradient and is confident the gradient field has sufficiently been established, it broadcasts its own "pathfinder" capsule that climbs the gradient-field to reach the initiator mote, leaving bread-crumb capsules as it goes. Although there is a theoretical minimal set of nodes capable of delivering a message along the path, assuming messages are not lost, we use more nodes than is strictly necessary because, in practice, messages may be lost.

Routed messages are not intended to be used for high-traffic message situations. Bill Butera describes an efficient implementation of communication buses on a paintable computing (or amorphous computing) substrate if such functionality is desired.

3.2.4 Promise

The proposed initiation of a promise operates occurs as follows:

- Dispatch our payload via routed message to the target. In this case, the payload is a capsule that will invoke a method-call on the target capsule when it arrives to initiation calculation. The payload will also periodically send messages back to the the promise initiator to let it know that it is still alive.
  - Implicit success: in other words, the routed message doesn't respond that it couldn't find the target. Proceed to the next stage.
  - Failure: the routed message was unable to locate the target capsule. This means that the code capsule in question and the cell that it is part of need to be regenerated. (We must regenerate the whole cell because the inability to find a given member of a cell means that the entire cell is inaccessible to us.)

    * We send a routed message to a code repository by targeting the dedicated regeneration capsule for the cell in question. The regeneration capsule is a capsule that serves as the point-man for a series of capsules with the names of all the capsules that make up the logic of a given cell.
    * We wait an arbitrary amount of time for the regeneration to occur, then restart from the beginning.

- We wait for a response capsule from the promise payload indicating that execution has completed, and potentially bringing capsules storing results with it.
  - We maintain a timeout counter while waiting. If we do not hear periodic messages from the payload indicating that it is still alive and that processing is therefore still occurring, we restart our execution. This is a reasonable action because the promise payload is using a routed message which will attempt to establish a new path to us if the nodes that comprise the existing path are disabled.
- We receive the promise result, and invoke a method call on a local capsule as appropriate to let it know that execution has completed.

3.2.5 Checkpoint

Checkpoints store data capsules sent to them by logic cells periodically during the process of a long-running completion, or when important results have completed execution. Each checkpoint has a set of logic cells whose data they are assigned to store. Multiple checkpoints may be assigned to handle the data for a given logic cell. However, logic cells establish a routed-message link with only one checkpoint at a time. This is accomplished by having master and slave checkpoints for a given logic cell. There may be only one master checkpoint for a given logic cell, but multiple slaves. Only the master cell is allowed to have an active version of the code capsule with the unique-id that the logic cell will attempt to establish a routed message path to. The rest of the cells will disable their master cell and use their slave logic until such time as the master disappears. At that time, the checkpoints will stage a new leader election among themselves to determine who will be the new master for the given logic cell.

All checkpoints for a given logic cell replicate the same state. The master checkpoint is responsible for propagating any new data as well as any deletions to the slave checkpoints. Master and slave checkpoints communicate routed message paths that are biased to avoid active logic nodes. This is the motivation for not having a logic cell contact all the checkpoints that are assigned to store its data. If the logic cell established routed message paths to all checkpoints, there would be an enormous amount of communications traffic that would very likely interfere with active processing nodes. By using only one routed message path to connect to the master checkpoint, we minimize the traffic running through potentially busy corridors. The master checkpoint can then use roundabout routed message paths that avoid high traffic areas to update the other checkpoints without significantly interfering with active processing.

When it comes time to actually restore state from a checkpoint, a request is initiated by the "regeneration capsule" in the code repository that is responsible for restoring the given logic cell to operation. The regeneration capsule contacts the master checkpoint via routed message, and requests the latest state snapshot. The acting master capsule for the given logic cell pulls the appropriate strings with the capsules in the checkpoint and sends the data along the routed message path to the newly restored cell. The logic cell in cooperation with the regeneration capsule and helpers then consumes the state and restarts execution of the logic cell.

3.2.6 Code Propagation

Code propagation is responsible for moving the code assigned to a cell to that cell, from the base-station or from a code repository. The mechanics are actually fairly simple. From the RGLL example, we already know how to provide a "codecast" module that propagates code throughout the network in a global fashion. We clearly need slightly more sophisticated logic than just global broadcast. The solution is that we use routed message logic to lay down a trail of bread crumbs from the base-station or code-repository to the target cell. We then use a modified codecast that follows the trail of bread crumbs to the target. Once the code reaches the cell, additional codecast logic ensures that all members of the cell have all the code capsules. Then, a capsule is released that follows the bread crumbs back to the code source, killing the codecast capsules and their payload capsules as it goes. In order to ensure that payload capsules didn't accidentally arrive on unintended capsules within proximity of the bread-crumb path, the kill capsule spreads from the bread-crumb trail to a limited depth. This is accomplished by giving the capsules a counter. For example, to deal with capsules within an additional 2 hops, the capsules could survive for up to 2 hops without seeing a bread crumb. After that, they die on arrival. The killer capsules occupy this region of the network to ensure that a steady-state is reached, and then purge themselves after a short period of time.

It should be noted that in order for the codecast to be successful and not cause accidental side effects, the capsules must be shipped in "stasis". This means that the *init* code of the capsules should not be activated as they are being pushed about. This can be accomplished through the use of functors, but given that this would be a fairly common problem, it would probably merit changing RGLL to add an enabled bit on capsules. This also provides us some additional flexibility. For example, to deal with the case where an inactive capsule arrives on a node that already has an active instance of the capsule, we can define that inactive capsules are not equivalent to their active brethren. This allows our inactive capsule to be moved around with its state intact, without affecting an existing active capsule or itself being affected. Once the capsule reaches its destination, we define the activation stage to be equivalent to the message being received. This provides a simple and consistent mechanism for dealing with collisions as well. Of course, these additional semantics could still be implemented through use of functors. All that would be required is that when we introduce a variable that defines us as active or not, we mark it as naming. When we wanted to make the capsule active, we would specialize it once more, which has the standard received message semantics.

3.2.7 Code Repository

Interaction with promises is slightly more complicated than you would intuitively expect. When promises use routed messages to attempt to locate a given code capsule, they use a broadcast search. This broadcast may stumble upon a code repository. Clearly, we don't want the promise to try and use the code capsule in the code repository; it's not set up to be executed and potentially has none of the desired state. Just like capsules being moved by codecast, we desire capsules being stored in a code repository to be disabled, and furthermore, invisible to routed messages. We therefore assume the invalid bit and semantics defined in code propagation are available to us.

A code repository is initialized by propagating a "gateway" capsule to the given code repository. The leader of the cell takes on the tasks of the gateway capsule. A routed message path is established between the base-station code capsule injection point and the gateway capsule. As capsules are received from the base-station, the gateway divvies them up among the members of the cell. Capsules are randomly distributed among the nodes in the cell, and duplicated to the maximum extent possible given the maximum storage capabilities of the cell. Although it is possible that more than once code repository is required to hold the sum total of the capsules that comprise a compiled RSEAM program, that is not the problem of the gateway capsule. Rather, the base-station is responsible for taking the set of cells allocated to be code repositories, and distributing the capsule load among them. Since cells are the basis of any real functionality, if any capsule that makes up part of a given cell's logic is present on a code repository, then all of the capsules that make up that cell's logic will be present.

When requested, code repositories can cause dead/missing cells to be reborn by locating an available cell and propagating the required code to the selected cell from the code repository via codecast. Code repositories always contain all of the capsules that make up a given cell, simplifying the problem. There is a particular capsule responsible for regenerating each logic cell, known as the “regeneration capsule”. This capsule serves as the contact-point via routed message for promises when they find that a cell is dead, as well as being the architect of the re-establishment of a given cell. Their final act is to request that the checkpoint provide it with the latest state available for the given logic cell.

3.3 Meta-Operations

Cellularization

Since cells are such a key part of an RSEAM implementation, it’s important to provide an easy way to take code designed to logically operate on a single computing node, and transform it to operate on a cell in a reliable fashion. This is achieved by using a functor-type method to transform the node-level capsules so that they have an additional activity boolean that inhibits their behavior unless they are a leader. These are tied into the cell-maintenance code; if the leader of the cell dies, a new cell leader will be elected, and the code capsules in question will be activated on that node. The non-leader nodes in a cell are quiescent save for relaying messages, and processing data in parallel when explicitly told to do so by their leader.

Parallelization

One of the clear benefits of cells is that there is great potential for parallelization. However, there are a limited number of parallel execution models suitable to the limitations of our communication bandwidth and per-node memory. A scatter/gather model driven by the cell leader where the cell leader stores all values, sends them out to the other members of the cell for processing, and then receives them again once the processing is completed is not workable; there is too much communication traffic. Our solution is to adapt a vector processing model, where every node is a “lane”, and the leader is responsible for driving execution. This means that all nodes keep the data that they process locally. When the results need to be shuttled somewhere else, possibly another cell, for further processing, a protocol routes the packets from each cell along a routed message path to the target. The capsule setup would be that the data inputs would be environmental ancestors of the processing blocks. This allows streaming processing to occur in an intuitive fashion.

For example, each capsule could hold M data points. Assuming we have N processors in the cell, this means that we process NM elements at a time. Once we process the data points, we can start transmitting the results to the next destination. By having multiple capsules of M data points on a node, we can effectively pipeline our implementation, especially if the routed-message path bringing new data in to the cell does not intersect the routed-message path sending data to the next destination.

Implementing this using functors is a reasonable task. Because we are using vector processing semantics, all we need is a block of code that takes some number of inputs and outputs some other number of outputs. For each different configuration of inputs and outputs we create a functor. The functor is responsible for iterating over the contents of the input data cells, invoking the provided code, and writing the result to the result data cells. The implementation for streaming data in and out of a cell is likewise implemented using functors, and potentially building on the cellularization functors.

Chapter 4

Conclusions and Future Work

4.1 Capsule Size

It remains to be seen whether effectively sized capsules in terms of code and data will be sufficiently small in a binary implementation to be practical. There is a limit to how small you can make a capsule and still have it be effective; once the capsules get too small, too much overhead will be spent on coordinating capsules to accomplish anything. There is also an upper-bound on the practical size of capsules, especially when compared to less extreme viral-computing models like Paintable Computing, or wholly data-passing models such as TinyOS. Larger capsules/communication packets utilize more of the available communications bandwidth and are more likely to be corrupted in-flight and therefore need to be discarded. On the upside, there is a force that pushes the optimal communication packet size upwards as communication speeds increase. The potential problem in all of this is that the minimum effective capsule size might be larger than the maximum practical capsule/communication packet size.

Unfortunately, the actual size of capsules can't be known until a compiler for RGLL is created that targets a real, binary platform such as the Berkeley Motes. There are many optimizations that can be performed, but we won't know how any of them pays off until the compiler is implemented. Part of the determination of whether the capsule sizes are practical depends on what the average communication bandwidth for sensor networks turns out to be. For example, the just recently deprecated line of Berkeley Motes, the Mica1, had a maximum communications bandwidth of 19.2kbps and operates using a single communications channel. Joshua Lifton's Push-pin Computers [Lif02] are capable of communicating at 166kbps on as single communications channel. There have also been reports of groups augmenting the Berkeley hardware designs with bluetooth transceivers, which offer much greater communications capabilities. Those who design future sensor network computing nodes will no doubt compromise between bandwidth and power utilization. What is clear is that the bandwidth will be greater than the meager 19.2kbps provided by the Mica1's, but it is not clear how much faster they will be.

4.2 Viral Computing Programming Model vs. Viral Computing Implementation

One of the important questions about RGLL that has yet to be answered is whether a viral computing implementation provide significant advantages when compared to a more traditional implementation that adopts the viral computing programming model. Butera's Paintable Computing model is an example of a relatively traditional implementation augmented to allow programs to be written in a viral computing model. The nuts and bolts that underly the conceptual viral computing model are all procedural. Communication between program fragments in the Paintable Computing model is higher level than explicit packet sends and receives, but is still just data.

In contrast, as described RGLL is a viral computing model with a viral computing implementation. RGLL's implementation could be redefined so that code and data are transmitted separately, moving the underlying implementation closer to Paintable Computing, and further from a viral computing implementation. In such a scheme, data tagged with the unique capsule id would be sent from capsule to capsule. The code portion of capsules would be cached. If a node received a data capsule broadcast for a code capsule that it did not yet posses, the node would issue a request to have the code itself broadcast. This change would place much more importance on the code layer supporting the capsules for proper operation, but from the perspective of the capsules, the execution semantics would be roughly equivalent. By not throwing code about with every broadcast, capsule code sizes could be dramatically increased. Investigation of such an implementation would prove quite interesting.

Another direction for future research is leveraging the fact that the currently defined implementation broadcasts code with data by allowing self-modifying code and dynamic capsule generation by other capsules. This takes a viral computing implementation as far as it goes. Such an implementation would allow radically new semantics. However, it is not clear that such functionality would be useful outside of the space where genetic algorithms are applicable. And in the bulk of those cases where genetic algorithms can be used, configuration via data remains an equally valid method, if potentially slower.

4.3 RSEAM

The most interesting possible future work stemming from the progress thus far is the full development and implementation of RSEAM. Coupled with an actual hardware target instead of pure simulation, the potential results would be very rewarding. It is certainly not a trivial undertaking. Although we have strived to provide a road that leads towards RSEAM, no one can know how long the road will be, nor what twists and turns exist until they try and walk to the end of it.

Bibliography

[AAC$^+$00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Embedding the Internet: Amorphous computing. 43(5):74–82, May 2000.

[Bac03] Jonathan Bachrach. *GOO Reference Manual*. MIT Artificial Intelligence Laboratory, http://www.ai.mit.edu/ jrb/goo/manual/goomanual.html, v44 edition, 16 January 2003.

[BN02] Jake Beal and Ryan Newton. Amorphous Infrastructure for Language Implementation. Final Paper for MIT Class 6.978: Biologically Motivated Programming Technology for Robust Systems, 10 December 2002.

[Bro02] Rodney Brooks. Creature Language. November 2002.

[But02] William Joseph Butera. *Programming a Paintable Computer*. PhD thesis, Massachusetts Institute of Technology, February 2002.

[CHB$^+$01] David Culler, Jason Hill, Philip Buonadonna, Robert Szewcz yk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT*, October 2001.

[CNW97] Daniel Coore, Radhika Nagpal, and Ron Weiss. Paradigms for structure in an amorphous computer. Technical Report AIM-1614, MIT Artificial Intelligence Laboratory, 6, 1997.

[GLvB$^+$03] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded system s. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, June 2003.

[IRW01] Michel Ingham, Robert Ragno, and Brian C. Williams. A Reactive Model-based Programming Language for Robotic Space Explorers. St-Hubert, Canada, June 2001.

[Lif02] Joshua Harlan Lifton. Pushpin Computing: a Platform for Distributed Sensor Networks. Master's thesis, Massachusetts Institute of Technology, 9 August 2002.

[McL99] James D. McLurkin. Algorithms for Distributed Sensor Networks. Master's thesis, University of California at Berkeley, Berkeley Sensor and Actuator Center, 16 December 1999.

[Nag99] Radhika Nagpal. Organizing a Global Coordinate System from Local Information on an Amorphous Computer. Technical Report AIM-1666, MIT Artificial Intelligence Laboratory, 12 August 1999.

[NSB03] Radhika Nagpal, Howard Shrobe, and Jonathan Bachrach. Organizing a global coordinate system from local information on an ad hoc sensor network. In *Information Processing in Sensor Networks*, 2003.

[VL02] Dimitris Vyzovitis and Andrew Lippman. MAST: A Dynamic Language for Programmable Networks. Technical report, MIT Media Laboratory, 31 May 2002.