Programming the Swarm

David Evans
University of Virginia, Department of Computer Science
July 24, 2000

Computing is rapidly moving away from traditional computers. Of the 8 billion computing units will be deployed worldwide this year, only 150 million are stand-alone computers [Tennenhouse2000]. Many programs in the future will run on collections of mobile processors interacting with the physical world and communicating over dynamic, ad hoc networks. We can view such a collection of devices as a *swarm*. As with natural swarms, such as a school of fish or an ant colony, the behavior of the swarm emerges from the simple behaviors of its individual members. The swarm is resilient to a small fraction of misbehaving members and to changes in the environment.

A fundamental challenge for computer scientists over the next decade is to develop methods for creating, understanding and validating properties of programs executing on swarms of communicating computational units. The key difference between swarm programming and traditional programming is that the behavior of a swarm program emerges from the total behavior of its individual computing units. Unlike traditional distributed programming, swarm programs must deal with uncertain environments and mobile nodes. The scientific challenges of swarm computing are to understand how to create programs for individual units based on the desired behavior, and conversely, how to predict high-level behavior based on the programs running on the individual computing units.

Recently, the building blocks for swarm computing have come into place. Advances in microelectomechanical systems and wireless networking have led to tiny, inexpensive devices with significant computing and communicating capabilities. Protocols for naming and communicating among disparate devices are emerging. Although there are many issues to resolve, many of the technological challenges have already been solved and there is an active and growing research community working on the devices and networking protocols that will provide the infrastructure for swarm computing.

We are a long way, however, from solving the deeper problem of how to program swarms of computing devices. My work will focus on creating and reasoning about swarm programs. One promising approach is the Amorphous Computing project at MIT [Abelson2000]. They have demonstrated the ability to produce complex structures (e.g., CMOS circuit topologies) in a computing medium consisting of scattered particles with limited memory, computation and communication capabilities. Instead of scattered particle environments, we will consider environments where the devices move and interact with their environment. The main novelty of my work will be the application of formal methods and lightweight mechanical analyses to swarm programs. This will involve techniques for describing both the desired behavior of a swarm program and the constraints on acceptable behavior in extreme environments. Since proving properties in general is infeasible, we will use lightweight mechanical analyses to approximate checking. This sacrifices soundness and completeness, but enables certain classes of properties to be checked efficiently and with limited programmer effort. In addition, we will develop techniques for construction swarm programs. One approach is to develop primitives and explore different mechanisms for composing swarm programs. A second approach will be to synthesize unit programs based on descriptions of the target environment and desired behavior. By separating functionality from resource use constraints and environmental considerations, this will support the development of versatile swarm programs. We will evaluate this work both analytically and by using a simulator to conduct experiments on the behavior of swarm programs.

1. Scientific Objectives

The scientific objectives of this research are to improve our ability to build and understand swarm programs. Swarm programs execute on scalable, ad hoc networks of unreliable devices that interact with the physical world. There are many problems to solve before achieving this goal, and my plan is to combine an incremental approach building on prior work with a more speculative approach.

Typical computation environments are undergoing a revolution. Computational devices are beginning to interact with the physical world. Tiny, inexpensive devices can be built that have considerable computing and communication capabilities. Despite these dramatic changes, the way we create software has undergone only minor and evolutionary change over the past forty years. The 1950s and 1960s were characterized by "programming in the small". Researchers developed languages and techniques for producing and understanding small, self-contained programs. From the late-60s until today, much programming language and software engineering research focused on "programming in the large". Techniques such as modularity, exception handling and inheritance were developed to improve our ability to build and understand large, more complex programs. These approaches are failing to keep up with the challenges faced by modern computing environments. My work strives to develop a new programming paradigm that is suitable for today's and tomorrow's computing environments: "programming the swarm".

Swarm computing differs from traditional programming in several ways. It assumes a computing infrastructure of many cooperating devices, each with limited ability to perform computation, to maintain state, and to communicate with its neighbors. Whereas in traditional computing, time and memory are the most limited resources, in swarm computing the most limited resource may be energy. Swarm programs must be able to adaptively tradeoff power consumption with computation and communication. In addition, they must be able to tradeoff resource consumption and behavior fidelity. For example, in many applications it is better to produce an approximate answer using 10% of the power required to produce the fully correct behavior than to produce the precisely correct answer.

This leads to a number of interesting research questions:

1) How can we specify programs in a way that encodes acceptable approximations for their functional behavior? Unlike traditional specifications, which provide precise postconditions given satisfied preconditions, specifications for swarm programs need to provide characterizations of an acceptable answer along with directives that express desirable tradeoffs between resource usage and precision. It is important to specify the range of acceptable behavior formally, so that analyses can establish correctness properties of implementations of swarm programs. These correctness properties will depend on a model of a physical environment.
2) How can we model devices and environments? In traditional program analyses, it is customary to assume all hardware components behave perfectly and that the containing environment behaves in well-understood and predictable ways. Work on fault-tolerance has explored a more realistic environment model, and succeeded in proving properties about systems allowing for a single (or limited multiple number) of failures. In swarm computing, we are dealing with the physical world. Programs need to be resilient to a wide variety of unpredictable events, without programmers needing to write code to explicitly handle every contingency. Resiliency does not mean they must continue to perform according to a precise functional specification, as it is clearly not possible to do so. The goal is to design systems that perform reasonably under adverse conditions, and perform without causing harm under catastrophic conditions.
3) What are the right primitives for swarm computing? The primitive control structures and statements from tradition computing cannot be mapped directly to swarm computing. Even simple concepts like a procedure call and an assignment do not have obvious analogs in a swarm computing environment. Primitives and libraries in a swarm computing environment need to capture common behaviors in a principled way. In particular, primitives should support programs that adapt to limited resources in a natural and efficient way.
4) What are the right mechanisms for combining primitives in a swarm program? Traditional composition techniques are not appropriate for a swarm environment as they depend on impractical amount of coordination among swarm members. Instead, we will seek to discover low-overhead composition mechanisms that still permit reasoning about the behavior of the combined program.
5) What is the right level of abstraction for programming in a swarm environment? Swarm computing programs should be written without the need to explicitly program individual devices. Programmers need to be able to describe the interesting and novel aspects of the high-level behavior they desire without needing to explicitly program the behavior of individual devices. It should be possible to describe the acceptable tradeoffs between functional behavior and resource usage without cluttering the code with resource management issues.

6) How can we test and validate swarm programs? It will often not be possible to test a swarm program in the field under realistic conditions, and in particular, under the range of possible environments it may face. A combination of static analysis and simulation offers the best hopes for success. In both cases, mechanisms to describe the range of acceptable behaviors of the program, and to specify the range of possible environmental conditions are essential.

We are many years away from determining the right answers to these questions. However, we are optimistic that the research program outlined here will provide the first steps towards these goals and lead towards eventual solutions. Good answers to the first two questions are necessary before we can hope to satisfactorily address the final four questions. As we develop ways to specify the behavior of swarm programs and model environments, I will develop methods and tools for reasoning about swarm programs. These will lead to the understanding of the primitives and combination mechanisms needed to develop a programming paradigm suitable for swarm computing. This programming paradigm will be designed in conjunction with tools for analyzing swarm programs and for synthesizing device programs based on high-level behavior and environment descriptions.

2. Impact

The potential applications of swarm computing are limitless. A few examples include:

Exploration. Imagine dropping an army of small robots from an airplane (or spacecraft on Mars). The goal is to explore a geographic area and send as much useful information back to a host as can be done with the limited energy available. A satisfactory program will need to support adaptable cooperation – for example, when one robot finds an interesting area (based on configured properties) but does not have sufficient energy to explore it completely, it should be able to communicate with nearby robots and attract them to the area of interest. Conversely, if one robot is destroyed, nearby robots should change their behavior to cover the area appropriately. Further, information obtained by the robots is of no use unless it can be transmitted back to the central controller. A result of swarm computing should be the ability to program exploration missions based on describing properties of interesting areas and tradeoffs between detail and scope. Instead of attempting to manually encode all these decisions into the behavior of individual devices, the individual programs should be generated mechanically based on a formal description of the desired behavior.

Search and Rescue (or Destroy). A variation on exploration is a more directed search for a particular item. Examples include searching for and rescuing skiers lost in an avalanche, locating precious minerals on the ocean floor, recovering a black box after a plane crash and destroying toxic waste after a storage leak. A swarm program could direct a collection of devices to cover an area. As sensors indicate a higher probability of being near the goal, additional devices can be attracted to a particular area. Once the target is located, devices cooperate to take the necessary action or pool resources to send a message back to the control center.

Sensor Networks. Many of the same problems with exploration are present in sensor networks. The difference is sensor networks are primarily interested in dynamic properties of an environment, and it is often reasonable to assume that individual sensors are at fixed locations. A sensor network around a battlefield or a toxic spill may be created by dropping a large number of sensors out of a helicopter. The sensors need to organize themselves into a useful network. A primary concern is minimizing the amount of redundant information transmitted, since this would waste the limited power available. A swarm computing program for a sensor network should be able to describe the desired information to be gathered, and the expected properties and extremes of the environment, and lead to automatic production of the individual device programs. In addition, this program must be resilient to individual devices failures.

Electronic Commerce. In the near future, people may have millions of agents authorized to act of their behalf. These agents may buy and sell information and goods on behalf of individuals or organizations. The relevance to swarm computing is that the owner of the agents must be able to define a policy that limits what the agents may do. It is not sufficient for policies to be defined for individual agents, the owner must be able to define and rely on a universal policy that constrains the aggregate behavior of the swarm of agents.

3. Prior Research

My research to date has focused in two areas: lightweight formal methods and policy-directed code safety. This section describes previous work in those areas and my plans to extend this work towards swarm computing.

3.1 Lightweight Formal Methods

There is a huge gap between the amount of effort and expertise required to use traditional development tools (such as compilers, integrated development environments, and test scripts) and formal techniques such as specifications, model checking, and program verification. On the other hand, a large class of common programming errors can be detected using simpler techniques. My research on lightweight formal methods explores what can be done by requiring only minimal programmer effort and without substantial changes to traditional development processes. This means lightweight formal methods must be supported by tools that require no user interaction and that run about as fast as a typical compiler.

To explore this space, I developed LCLint, a tool for statically checking C programs [Evans94, Evans96, Evans2000a]. LCLint provides a first step towards adoption of formal techniques and mechanical analysis. If minimal effort is invested adding annotations to programs, LCLint can perform stronger checks than can be done by any compiler or standard lint. LCLint checking ensures that there is a clear and commensurate payoff for any effort spent adding annotations. Some of the problems that can be detected by LCLint include: violations of information hiding; inconsistent modifications of caller-visible state; inconsistent uses of global variables; memory management errors including uses of dead storage and memory leaks; and undefined program behavior. LCLint checking is done using simple dataflow analyses; it is as fast as a compiler and requires no user interaction.

The performance and usability goals for LCLint require certain compromises to be made regarding the checking. In particular, we sacrifice soundness and completeness. While this would not be acceptable in many situations, it is a desirable tradeoff in a typical industrial development environment where efficient detection of program bugs is the overriding goal. A small number of false positives and false negatives are preferable to excluding more ambitious checking or requiring and untenable burden on users. LCLint is used by more than a thousand sites in industry and academia and has been the subject or articles in leading popular journals (Linux Journal [Orcero2000], Linux Gazette [Pramode2000]).

Extending LCLint. Recent work on annotation-assisted static checking is progressing on two fronts: detecting buffer overflow errors and providing support for user-defined checking and annotations. This work is fund by a NASA grant (joint with John Knight) on practical use of formal techniques in aerospace software systems. This work will increase our understanding of lightweight static checking techniques and increase the power of LCLint in ways that will enable its use in a swarm computing environment. In particular, the work on buffer overflows will require the development of heuristic techniques for modeling computations more deeply than is currently done by LCLint. The work on user-defined checking will provide a useful framework for experimenting with different constraints we may wish to check on swarm programs. It allows programmers to invent new annotations, express syntactic constraints on their usage, and define checking associated with the annotation. Annotations introduce state that is associated with both declarations and intermediate expressions along symbolic execution paths. The meaning of an annotation is defined by semantic rules similar to typing judgments, except they may describe more flexible constraints and transitions than is usually done with typing judgments. We are defining a general meta-annotation language that can define a class of annotations in a simple and general way. Meta-annotations define constraints and transition functions when storage is assigned, passed as parameters, returned from a function, and when control enters or exits a block or function. By developing a framework for easily introducing new checks, we will be able to experiment with a variety of different checks that may be useful in swarm programs.

Towards Swarm Computing. LCLint's current analyses assume a program is executing sequentially on a single processor. To check useful properties about swarm programs, we need to develop analyses that assume a much different environment – many independent devices executing programs asynchronously and unreliably. Checking properties of swarm programs will depend on developing heuristics for communicating devices analogous to the heuristics developed to understand loops in sequential programs. The goal is to be able to determine properties about a swarm of devices from a static analysis of individual device programs and a description of the range of execution environments.

The kinds of properties we can realistically imagine checking using lightweight mechanical analysis of swarm programs concern resource usage and behavior extremes. The goal is not to prove that a swarm program behaves correctly in all environments, since it is highly unlikely that this could be done. Instead, we will seek to develop tools and techniques for detecting likely problems in swarm programs and alerting programmers to things that might go wrong.

3.2 Policy-Directed Code Safety

I have developed Naccio, a general architecture for code safety that addresses two weaknesses in current code safety systems [Evans99a, Evans99b]. One weakness is that traditional code safety systems cannot enforce policies with sufficient precision. For example, suppose a user wished to enforce a constraint on the amount of bandwidth untrusted programs may consume. Existing code safety systems, such as the Java sandbox, cannot enforce such a policy without denying network use altogether since there are no safety checks associated with sending or receiving data. The problem is more fundamental than just that the designers made arbitrary choices about which safety checks should be provided. They were hamstrung into providing only a limited number of checks by a design that incurs the cost of a safety check regardless of whether it matters to the policy in effect. Naccio enforces safety policies in a way such that safety checks can be associated with any resource manipulation, yet the costs of a safety check are incurred only when the check is relevant to the effective safety policy. This is accomplished by using static analysis to eliminate unnecessary checking before the policy-enforcing program is executed.

Another problem with current code safety systems is that the means used to express policies is *ad hoc* and platform-specific. The author of a safety policy needs to know low-level details about a particular platform and once a safety policy has been developed and tested it cannot be transferred to a different platform. Policies should be selected based on the level of trust and type of application, not based on the execution platform. The Naccio architecture provides a platform-independent way of defining and enforcing safety policies in terms of abstract resource manipulations. Safety policies are defined using code fragments that account for and constrain resource manipulations. Resources are described using abstract objects with operations that correspond to manipulations of the corresponding physical or logical resource. Naccio enforces a policy by generating a policy-specific version of the relevant parts of the platform library, and by transforming programs to use the policy-specific library. A platform interface provides an operational specification of how system calls affect resources. This enables safety policies to be described in a platform-independent way and isolates most of the complexity of the system. The Naccio architecture supports tradeoffs between the time required to generate a safety policy, the time required to transform an application and the run-time costs associated with enforcing a safety policy.

We have built two implementations of Naccio: Naccio/JavaVM that enforces policies on JavaVM programs [Evans99b]; and Naccio/Win32 that enforces policies on Win32 executables [Twyman99]. Results from experiments with the prototypes indicate that it is possible to support a large class of policies without sacrificing performance for simple policies. Naccio is designed to make the run-time overhead minimal and ensure that the run-time costs associated with enforcing a safety policy are directly related to the complexity and ubiquity of the policy.

Towards Swarm Computing. Work to date on Naccio has focused on enforcing a policy on a single execution of a single program. In a swarm computing environment, we need to define and enforce policies that apply to a collection of devices executing possibly different programs. Both the definition and enforcement of policies over collections of devices pose substantial research challenges. In most cases, it will be reasonable to assume that swarm programs are not deliberately hostile. This means our efforts to enforce and describe safety policies should be directed towards preventing unexpected interactions from producing harmful consequences, rather than the traditional security concerns of preventing a clever and motivated attacker from exploiting a system vulnerability.

The Naccio approach of describing a policy in terms of constraints on abstract resource manipulations maps well to a swarm computing environment. The difference is that resources are harder to monitor and mediate in a swarm computing environment than those constrained by traditional policies since they are spread over many devices. For example, consider a policy that constrains the total bandwidth that may be used by a swarm program. In the single-program situation we can enforce such a policy by monitoring the bandwidth used by a single execution and issue a warning (or alter the behavior) when the prescribed limit is reached. We could enforce a resource constraint policy on a swarm program by dividing the resource limit among all nodes and relying on individual nodes to enforce their constraint. This does not make maximal use of the limited resource since some nodes will not need to use their entire allotment, but are unable to redistribute resource privileges to other nodes. More troubling, one defective or renegade node can violate its individual constraint without detection. Another straw man approach would be to have all nodes communicate with a central authority that monitors cumulative resource usage and can disable misbehaving nodes before they cause substantial harm. Relying on a single point of failure is contrary to the realities of swarm computing. A satisfactory solution is likely to involve a combination of local monitoring and enforcement, and distributed nodes that detect defective or renegade nodes while supporting resource distribution across local boundaries.

Since Naccio works by transforming programs, it is possible to separate a resource usage policy from the main programming task. This offers the possibility to create the unit programs independently of a high-level policy. Unit programs can be created with a focus on the desired functionality, while non-functional properties like resource consumption are defined by a safety policy. This allows changes in functionality to be decoupled from other considerations, and will make it easier to create swarm programs that adapt to changing environments with different requirements. This will draw from our work on aspect-oriented security [Viega2000].

4. Research Plan

My research plan will be directed towards making progress on the scientific objectives described in Section 1. Initial efforts will be directed at gaining an intuitive understanding of swarm programs and building a library of swarm programs and environments. After that, research will focus on developing methods for describing and reasoning about swarm programs, in conjunction with techniques for constructing swarm programs.

4.1 Incremental Research Directions

Work will continue on lightweight formal methods and policy-directed code safety, while I ramp up more speculative work on swarm computing. As described in Section 3, this work will lay the groundwork for my swarm computing research. In particular, the extensions to LCLint will improve our ability to perform approximate analyses on programs and provide the ability to easily define new properties to check. Work on using Naccio to define and enforce distributed safety policies will lead to a better understanding of how to describe non-functional specifications for swarm programs.

4.2 Experimental Swarm Programs

We will experiment with two types of swarm programs – primitives, which may be considered part of a swarm computing library but are not useful stand-alone applications; and applications, which compose and adapt primitives to produce a useful swarm program.

Some examples of swarm computing primitives include:

Disperse – devices spread out over an area. The end state satisfies a constraint of the form that no two devices are within *d* distance of each other. A naïve implementation of disperse would program each device to detect the number of devices within *d* of it by transmitting a message with the appropriate power to cover the *d* distance region surrounding it. If one or more devices respond, move randomly for a short distance and try again. If the number of nearby devices has increased, try moving in a different direction. Once no devices respond for a threshold number of trials, stop. It is not easy to determine the initial conditions for which the naïve disperse program terminates. Initially, we will explore these through simulations. Once a reasonable intuition is acquired, analytical approaches will be pursued. The naïve disperse program is neither time nor power efficient. Alternatives that trade-off time and power given different costs for communication, computation and movement will be explored. One possibility is to divide the area into larger regions first, and move devices between regions until each region satisfies the constraint scaled to its size. Then continue to divide the regions and disperse until the final constraint is satisfied.

Generalized Disperse – the end state satisfies a constraint of the form that no more than *n* devices are within *d* distance of any device. Similar to two-device disperse, except that by dealing with multiple devices it leads to more flexible solutions and more complex analyses.

Clump – devices gather into clusters. The end state satisfies the constraint that almost all devices are within *d* distance of at least *n* devices. Devices send messages to find neighbors and move toward them, forming larger clusters of devices. Clusters can send stronger messages to look for nearby clusters, and merge to form super-clusters.

Attract – similar to clump, except devices are attracted to a particular location. A special device is initialized as the attractor and begins transmitting messages that indicate its location. Devices move towards this signal, while retransmitting it to attract more distant devices.

Swarm – devices move in a general direction. All devices move in a similar direction (but without colliding) towards a target region.

Scan – devices cover an area. At least one device touches every *r*-sized region of a specified area. The previous primitives can all be described solely in terms of constraints on the desired final state. A scan depends on a history of states, where for every *r*-sized region there exists a state such that at least one device is in the region. An inefficient implementation might use devices in one of three states – "searching" and "scanning" and "marker". Devices periodically transmit a message over an *r* distance indicating their presence and state. Each device begins in the searching state, and starts traveling East. If a device encounters another device in the searching state, it moves South one distance unit, and then continues to move East. If a device encounters another device in the scanning state, it moves South one unit, and then continues to move East. If a device encounters the region boundary while in the searching state, it moves North until it encounters a device in the marker state. If it reaches the top boundary, it enters the marker state. If it reaches a marker, it enters the scanning state and starts moving West. The marker devices moves South one unit. Similar techniques can be use at the Western boundary to ensure that the entire area is scanned. Obviously, this is not a terribly efficient scanning implementation assuming moving is more expensive than communication. It is also not resilient to device failures, since a marker failure will require substantial wasted effort. An alternative implementation might use disperse approaches to divide the area into regions, or have devices communicate information to other devices about where they have already scanned. If information about absolute location can be determined, or if devices can deposit pheromones as the scan, further efficiency improvements are possible.

Broadcast – transmit a message to (almost) all members of the swarm. This may only be possible given constraints on the configuration of the swarm, for example if a device is too far away to be reached it would be considered lost.

Partialcast – transmit a message to a specified fraction of swarm devices. A generalized partialcast should be able to reach between *x* and *y* percent of the devices in the swarm.

Directed message – send a message across the swarm in a particular direction. Here, it doesn't matter how many devices receive the message, as long as enough devices in the desired region receive it.

For each of the primitives, there are a wide variety of implementations that trade off resilience (how well do they match the desired functionality given a wide range of environments and device failures), efficiency (the amount of resources they consume), and speed (how quickly do they achieve an approximation of the desired behavior). A swarm computing library will need to include several implementations of each primitive, along with descriptions of their non-functional properties. A swarm program will need to select and compose primitives based on its desired functional and non-functional properties and a model of its execution environment.

Composition. In traditional computing, procedures are composed in a straightforward, sequential manner. The output state of one procedure is used as the input state of the next procedure. The behavioral specification of the composed procedure can be readily derived from the specifications of the two units. In swarm computing, composition is much more complicated. We will explore several different meanings of composition in swarm programs. Some possibilities include:

- Unit sequential composition – each device composes two programs in the traditional sequential way. Since devices are unsynchronized, however, this means some devices will be executing the first program while others have already begun executing the second program. In cases where the first program is guaranteed to eventually terminate, the final state should satisfy the constraints of the second program.
- Unit parallel composition – each device composes two programs in parallel. This could be mimicked on sequential devices by systematically switching execution between each program. For example, a motion and sensing task could be composed to produce a searching program.
- Divided composition – some fraction of devices execute the first program while the rest execute the second program. Imagine a divided composition of the clump and disperse programs.
- Global composition – the programs are composed over the swarm in a synchronized way. Global composition is expensive, since it requires a broadcast message that triggers each device to switch programs at approximately the same time.

The primitive and composition operators will be used to construct several sample swarm programs based on the applications described in Section 2. For example, we could construct a search and rescue application by a unit parallel composition of scanning and a sensing program that is unit composed with an attracting program that begins when the sensing program has detected the target. Biology provides an excellent source of example swarm programs. Seemingly lowly animals have evolved complex group behaviors that emerge when large numbers of individuals follow simple rules. One compelling example is the house-hunting behavior or honey bees. When a hive becomes too large, a portion of the hive will split off and journey to a nearby tree. This provides a temporary home, while members of the hive search for a good site for a permanent home. These scouts venture out (often several kilometers away from the temporary home) looking for a good site. A scout that finds a candidate location returns and performs a "waggle dance" that encodes the location of the site and how well it meets their requirements. Other bees in the hive may start mimicking the same dance. Once a consensus has been reached (that is, enough of the bees are dancing for the same location), the hive travels to its new home. The bee house-hunting example illustrates a number of swarm computing primitives and composition mechanisms – disperse (when the hive splits), swarm (when the split hive moves to its new location), a variation on scan (when the scouts search for a new site), region messages (when returning scouts perform their waggle dance), and swarm (when the hive moves to its permanent home).

A suite of sample swarm programs will be developed and used to conduct simulated experiments. Initial experiments will be conducted using the MS Modeling System, a simulation system developed at the University of Virginia by John Knight and his students. The MS Modeling System is capable of modeling large networks (up to 40,000 nodes) on desktop PCs. It is designed to support experiments on survivable systems and allows operators to inject large-scale faults into the simulated network. As such, it provides a useful platform for conducting swarm computing experiments. One challenge will be to develop a suitable model of physical environments. We will need to use descriptions of environmental conditions that may interact with swarm devices to enable simulations of the resiliency of a swarm program.

I do not expect to spend effort on building physical devices – there are many researchers actively working in this area. However, it will be useful to conduct experiments using physical devices in real environments. Although these experiments are harder to conduct than simulations, they provide a stronger basis for understanding than can be obtained from simulations alone. I expect within a few years, it will be possible to inexpensively acquire the physical devices needed to conduct swarm computing experiments in a physical environment.

4.3 Developing Swarm Specifications

In order to reason about swarm programs, we need a way to formally capture the intended behavior of a swarm program. Current specification techniques are inadequate for describing swarm programs. Several specification techniques have been developed to support mobile computation, including Mobile UNITY [Roman97], the Distributed Join-Calculus [Fournet96], and Mobile Ambients [Cardelli98]. Although these tools provide methods for modeling location and failure in specifications, they are all at the level of individual processors. This means they don't scale well enough to handle swarm computations involving thousands of individual devices. Instead, we need to develop specification techniques that capture the global behavior of a collection of interacting devices.

One approach under consideration is to view a swarm program execution as a sequence of global states, each capturing the state of each unit device. The state of a unit device includes its location, as well as any internal state information, and the program it is running. Since there is no time synchronization, the granularity of global states raises important questions. The state of a swarm computation may change at any time, and only infinitesimal granularity could capture the entire behavior. Analyses involving an infinite number of global states are infeasible. Instead, I propose to describe global state with a predicate that captures a range of similar states. The system remains in the same state until the predicate is no longer true. A swarm program is specified by a temporal logical predicate on a collection of global state predicates.

In addition to the functional specification captured by the global state predicates, a swarm program is also described by a non-functional specification that captures properties like resource use and resiliency to different environments. Keeping the non-functional specification separate from the functional specification allows us to develop a library of non-functional specifications that can be used interchangeably with functional specifications.

We will develop functional and non-functional specifications for the different primitive implementations derived for our experimental swarm programs. These will be produced by first producing informal specifications of the functional characteristics of each primitive. The informal specifications will be refined into global state sequence specifications. For most of the primitives, these contain only two states – the initial state and the final state. These states need to be defined in terms of an arbitrary number of devices. For example, the specification for disperse would use an unconstrained initial state. The final state would match any global state where no two devices are within *d* distance of each other. For action-oriented primitives like scan, the specification is an unordered collection of partially defined global states. Each state constraints one unit square of the area to contain a device, but places no constraints of the other devices. At a particular instant, the swarm state may match several of the stated in the specification collection. The specification is satisfied by an execution if each global state in the specification is matched by the actual state at some instant of the execution.

Specifying the non-functional properties of the primitive implementations will be done by analyzing their behavior under different conditions. The non-functional specifications will be described in terms of the properties of the devices and environment. The properties describe the resource consumption required for communication, computation and motion; the capabilities of the device such as memory, processing speed and message transmission; and the reliability of the device. In many cases, it will not be possible to analytically determine exact non-functional specifications. Instead, approximate specifications will be produced based on a combination of analysis and simulation results.

4.4 Developing Models

In order to conduct meaningful simulations of swarm programs, we need models of both the devices and their physical environment. The basic models will be derived from physics. Properties of power use and radio transmission are reasonably well understood. A variety of device models will be used in experiments based on realistic assumptions of available devices today and in the future. Failure rates can be set in the model and adjusted as a function of environmental conditions.

Modeling the environment is more interesting since our goal is to model the range of possible environmental conditions to which the swarm program may be subjected. In a physical environment, this can involve weather as well as disturbances by external devices and creatures. It is impossible to predict all possible environmental conditions. Nevertheless, we would like to be confident that a swarm program behaves reasonably under diverse conditions. I plan to develop a method for modeling environments and a library of different environments for swarm programs. These would be used initially in simulations, but eventually in analysis and synthesis of swarm programs also.

4.5 Analyzing Swarm Programs
Experimenting with swarm programs in simulations may provide some experience and insight into the behavior of swarm programs, but does not provide a satisfactory basis for reasoning about and validating swarm programs. Given the inherent unpredictability of swarm program executions – asynchronous operation, unreliable messages and devices, changing environments, etc. – a large number of test executions does not provide much confidence that the swarm program will behave reasonably in successive executions.

I will seek techniques for analyzing swarm programs in ways that increase confidence that executions will have certain properties. This will involve developing tools that take functional and non-functional swarm program specifications, implementations of device-level programs and models of devices and environments and use static analysis techniques to identify likely problems with the implementation. If most swarm programs are constructed by composing primitives, the main challenges are to reliably specify the behavior of the primitives and to understand the semantics of different composition mechanisms. Since swarm specifications are by nature approximations, the composition mechanisms will need to capture the range of possible behaviors of the composed program. With each composition, the behavior becomes less and less predictable. We will seek to develop combination rules that allow us to understand this effect and gain the ability to reason about complex swarm programs. The non-functional properties can be analyzed independently of the functional properties. The non-functional analysis can and must be more precise than the functional analysis since it is important that we can make well-supported claims about non-functional properties.

The initial analysis work will depend on functional and non-functional specifications of the primitives. This will be generated by hand, based on understanding the primitive programs and conducting simulator experiments. Any reasoning done using these specifications will be inherently weak, since our confidence in the correctness of the primitive specifications is low. In order to improve this, we will strive to develop techniques for reasoning about low-level swarm programs such as the primitive implementations. This requires the ability to describe the range of high-level behaviors possible from a description of the device programs and formal models of the devices and environment.

4.6 Synthesizing Swarm Programs
The development of primitives and analysis techniques will make it easier to construct swarm programs, but still leave the programmer with a challenging task. Implementing a high-level behavior in terms of individual device programs is a considerable challenge.

A first step towards synthesizing swarm programs is to write a swarm program in terms of a library of primitives and use device and environment specifications to select the appropriate implementation of a primitive to use. Since the primitive implementations have associated non-functional specifications, we can select appropriate primitive implementations based on the overall requirements. This depends on understanding the interactions among different programs under different composition mechanisms. Based on our analysis work, it should be possible to determine the effects of selecting different primitive implementations and select the optimal combination of primitive implementations for a particular program. This choice will depend on the device and environment models, as well as the non-functional requirements of the program.

A much more ambitious goal is to produce the device programs for a primitive swarm program from the high-level behavior description. It is not realistic to hope that we could completely automate the process. However, there is an opportunity to provide great assistance to programmers in an integrated development environment. This would involve a refinement process where a programmer starts by describing a desired high-level behavior. The development tools produce a swarm program based on this description, and present the programmer with an analysis of its functional and non-functional behavior. In an iterative process, the programmer would refine the high-level behavior description with hints to the development environment, until the resulting swarm program is satisfactory. As our analysis and synthesis techniques improve, the amount of interaction required to produce the desired swarm program should diminish.

5. Evaluation

The scientific objectives of this work are to improve our ability to build and understand programs that execute on a swarm of devices in a physical environment. Success will be measured by our ability to establish general principles for creating and reasoning about swarm programs. It would be foolhardy to suggest that all the problems of swarm computing will be solved within the next five years. Instead, we hope to move swarm computing from the where it is now (roughly equivalent to where sequential programming was in the days of Charles Babbage and Ada Lovelace) to where traditional programming was in the early 1960s. This means we will have a reasonable base of well-understood primitive operations, along with mechanisms for composing those operations. We will have techniques for reasoning about the behavior of simple swarm programs, and mechanisms for understanding the behavior of larger programs in terms of the behavior of their components.

Our evaluation will use experiments that focus on particular swarm computing problems, such as the "search and rescue" and mobile sensor network tasks described in Section 2. By developing different swarm programs targeted to particular tasks and environments, we will explore methods for creating and reasoning about swarm programs. The test of these methods is in how well they allow us to build swarm programs that behave predictably to achieve a desired goal. Analyzing the predictability of swarm programs will depend on both analytical techniques and simulations with variable environmental models.

The real test comes when the methods developed are applied to a new task and environment. If the methods are useful, it will be possible to develop a satisfactory program for the new task in substantially less time then was required for the earlier tasks. After a few task iterations, it should be clear whether the principles and methods discovered are general enough to be useful in swarm computing, or are limited to a particular scenario. Behaviors found in natural biological systems provide good test cases for swarm computing. The limits on computation and communication of insects, and the fact that group behaviors had to evolve in a natural environment, mean that we should be able to duplicate observed behaviors of swarms of insects with a swarm program. Group behaviors in nature range from the fairly simple pheromone-routing of ant colonies and tent-building of caterpillars, to the complex workings of a bee hive. Many of these present useful examples for swarm computing.

6. Background

Much work has been done on building the types of micro devices and wireless networks necessary to produce a swarm computing infrastructure. This section reviews enabling technologies and surveys work that may provide some of the building blocks needed to create swarm programs.

6.1 Enabling Technologies

Swarm computing depends on the availability of cheap, small devices capable of computation and communication. There is good cause for optimism that micro devices and wireless network will soon be commonplace.

Devices. The steady improvement of microprocessors has lead to devices costing a few dollars that are as capable as desktop computers of 1985. Recent advances have led to dramatic reductions in the cost, size and power consumption requirements for electronic devices, wireless communications, and microelectomechanical systems (MEMS). The Smart Dust project at Berkeley is building cubic millimeter-scale self-powered autonomous nodes with the ability to do computation and communication [Kahn99]. Eventually, we may be able to construct even smaller and cheaper devices using molecules. It is expected that prototype molecular electronic devices will be built within a few years, although simple computers are probably at least a decade away [Rotman2000]. The age of inexpensive computing and communicating devices is on the horizon. We can build the devices – the question is whether or not we can program them to do useful things.

Wireless Networking. Although the principles of swarm computing are not necessarily tied to wireless networks, most interesting swarm computing environments are wireless. Although researchers are working on wireless communication using chemicals and light, the most promising short-term technology is radio.

There are several efforts underway to produce standardized, low-power wireless networking protocols. These range from IEEE 802.11b [IEEE2000] for high-bandwidth wireless local-area networks to the Bluetooth [Bluetooth2000] standard for low-cost, low-power small area networking which is receiving substantial industry support. Progress in this area seems inevitable – the essential infrastructure is already available, what is lacking is the ability to exploit it in useful applications.

6.2 Naming

In traditional networking, nodes are named by addresses that encode their location. DNS [Mockapetris87] encodes a search path to a node in its name. Swarm computing deals with mobile ad-hoc networks where nodes are not tied to fixed locations or known addresses. Primitive network services, such as routing, naming and resource discovery, pose major challenges for ad hoc networks. In fact, swarm environments will not provide traditional versions of these services since swarm programs should not depend on addressing particular nodes. Instead, swarm programs will address groups of nodes by proximity or attributes.

One promising approach to naming is the Intentional Naming System (INS), a resource discovery and service location system designed to support dynamic networks [Adjie99]. Names are specified using a hierarchical structure of attribute-value pairs. Services periodically advertise their name specifiers to the system. *Intentional Name Resolvers* (INRs) cache these names with expiration times. A known entity in the system called the *Domain Space Resolver* (DSR) keeps a list of all available INRs and hosts that are not currently running INRs but could be used to run them. Messages can be routed to either the best service node that satisfies the intentional name (*intentional anycast*), or to all service nodes that satisfy the name (*intentional multicast).* Adapting INS to a swarm computing environment would involve removing the dependence on a central DSR. A swarm computing analog of intentional anycast and intentional multicast would propagate messages through a swarm in a goal-directed fashion. Individual nodes would need to decide whether or not to retransmit a request based on limited local knowledge including notions of proximity and resource consumption.

6.3 Routing

Wireless ad hoc networks introduce new challenges in routing. In a swarm computing environment, it is not clear whether node-to-node routing is even a provided network service. Messages may be sent in a general direction, or dispersed throughout a computing medium. In some cases, however, it will be important to send a message to a particular, distinguished node. For example, one or more nodes in a swarm computing environment may be tethered to a power source or wired network. In order to send messages outside the swarm, devices need to be able to route messages to the distinguished node. Messages will be sent with multiple-hops – the challenge is to determine which and how many nodes need to retransmit a message to achieve the desired probability of correct transmission with the minimum resource consumption. Designing multi-hop routing algorithms that minimize power consumption is or crucial importance – sending a message using n short hops consumes approximately $1/n^{th}$ the amount required to send it using a single long hop.

One novel approach to routing is inspired by the behavior ant colonies use to find the best route to a food source [Bonabeau2000]. An ant returning from the food source leaves a trail of pheromones along its path. Since this ant will return to the food source along the same trail, depositing more pheromone, the trail it follows will contain a higher density of pheromone that a longer path to the food source followed by a different ant. Ants looking for food will follow the trail with the highest density of pheromone. Since the pheromone decays with time, the path-finding technique is resilient to new obstacles and changes in the food supply. Biological mechanisms used by insect swarms provide a source of inspiration for swarm computing. It is easy to see computational analogs to the ant colony behavior where local decisions lead to globally desirable results. Nature is full of fascinating examples of swarm behavior such as honey bees searching for a new home [Camazine99] and beetle circle defenses [Waldbauer2000].

6.4 Sensor Networks

Instead of routing messages between specific nodes, work in sensor networks has explored protocols that transmit information to nodes as needed based on application constraints. A typical scenario is a collection of sensors is dropped on a battlefield to monitor troop movements and communicate relevant findings back to a command center. One technique is a communication paradigm known as *directed diffusion* that distributes messages in a data-centric way [Intanagonwiwat2000]. Sensor nodes produce data in the form of attribute-value pairs. A sensing task is disseminated through the sensor network by setting up gradients that draw data matching requested attributes towards the request origin. Rather than specify this node directly, however, directed diffusion sends data back in the direction of its request without needing to route data to a specific origin node. The use of gradients enables robust, power-efficient communication without the need for complex routing algorithms. In addition, their techniques take advantage of task-specific algorithms for aggregating sensor data. Subramanian and Katz have generalized the techniques to develop an architecture for building self-configurable systems where a large number of sensors automatically coordinate to achieve a sensing task [Subramanian2000]. Swarm computing depends on self-organizing systems. A swarm program will use known techniques for organizing nodes to produce a network suitable for a given program. One goal of this research will be to produce methods for self-organizing nodes based on the desired properties of the application, instead of requiring explicit programming.

6.5 Tuple Spaces

Linda introduced a new model for concurrent programming based on the notion of tuple spaces [Gelernter85]. A tuple space is a global shared associative memory. Tuple spaces provide primitives for adding a tuple to the space, reading the value of a tuple that matches a template in the space, and taking (removing and returning) a matching tuple. Both blocking and non-blocking versions of the read and take primitives may be provided. Since values in the tuple space cannot be modified, there is no need for synchronization – the values themselves provide all the necessary coordination. Further, nodes do not need addresses since messages are not sent to particular nodes but placed in the shared tuple space.

Several recent systems, most notably Sun's JavaSpaces [Freeman99] and IBM's T Spaces [Lehman98] have considered using tuple spaces to provide a middleware platform for linking devices. T Spaces combines a database with tuple spaces, to provide a collection of tuple spaces with different visibility and support for transactions. The tuple space approach has many desirable properties, but a number of challenges would need to be overcome before it could be used in a swarm computing environment. Scaling and mobility present major problems, since it is normally assumed that all nodes can see the entire tuple space. Partitioning a tuple space into neighborhoods may offer a partial solution, but this requires substantial changes to the programming model. One approach is explored by LIME [Picco99], a tuple space system designed for mobile environments. In LIME, tuple spaces are transiently shared and distributed across mobile hosts. Instead of providing a global tuple space model, the tuples visible to a particular node change dynamically according to network proximity. In a swarm computing environment, it will not be practical to maintain a traditional tuple space even if it is transient. Instead, there may be a use for an "ether" space where tuples are broadcast to neighboring nodes. Rather than maintaining a persistent, shared tuple space, the nodes receiving a tuple can either react to it, retransmit it to a wider area or ignore it. It is unclear whether or not the much weaker semantics can support a useful programming model, but I believe it will be worthwhile to explore weaker variations on tuple space semantics for a swarm computing environment.

6.6 Amorphous Computing

The Amorphous Computing project at MIT is developing techniques for programming self-assembling systems of unreliable, irregularly placed asynchronous computing elements [Abelson2000]. An amorphous computing medium is a collection of particles all running the same program with modest memory and computing power, spread out over a geographical space. One technique for programming an amorphous computing medium is the Growing Point Language (GPL) [Coore98]. GPL uses a botanical metaphor of growing points that can secrete pheromones that act as messages to nearby nodes, and lay down material that alters the state of particles in the amorphous computing medium. Coore has demonstrated that this model is sufficiently powerful to program complex structures such as the interconnect topology of a CMOS circuit. There are many similarities between swarm computing and amorphous computing. For swarm computing, I will assume somewhat more powerful nodes than is the case with amorphous computing, and focus on creating programs that involve motion and interaction with an environment. Further, my emphasis will be on techniques for describing and reasoning about the behavior of swarm programs.

7. Summary

Swarm computing is an exciting and challenging area, and a promising source of research problems for computer scientists for many years to come. Although the practical applications of swarm computing are in their infancy, there is great potential for useful applications. As the inevitable widespread availability of inexpensive computing and communicating devices fast approaches, the need for methods to program these devices is readily apparent. Successful swarm programming will depend on our ability to reason about swarm programs and construct device programs based on high-level goals. The research program outlined here provides the first steps towards that target.

References

[Abelson2000] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. *Amorphous Computing*. Communications of the ACM, Volume 43, Number 5, p. 74-83. May 2000.

[Adjie99] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. *The design and implementation of an intentional naming system.* Proceedings of the 17th ACM symposium on Operating systems principles. p. 186-201. December 1999.

[Bluetooth2000] *Bluetooth Forum*. http://www.bluetooth.net.

[Bonabeau2000] Eric Bonabeau and Guy Théraulaz. *Swarm Smarts*. Scientific American. p. 72-79. March 2000.

[Camazine99] Camazine, S. et al. *House Hunting by honey bee swarms.* Insectes Sociaux Volume 46 Number 99. p. 348-360. January 1999.

[Cardelli98] L. Cardelli and A. D. Gordon. *Mobile ambients*. In *Proceedings of FOSSACS'98, International Conference on Foundations of Software Science and Computation Structures*. Lecture Notes in Computer Science, Springer-Verlag. Volume 1378, p. 140-155. 1998.

[Coore98] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer.* MIT PhD Thesis. December 1998.

[Estrin99] Deborah Estrin, Ramesh Govindan, John Heigemann. *Scalable Coordination in Sensor Networks*. USC Technical Report 99-692.

[Evans94] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering. p. 87-96. December 1994.

[Evans96] David Evans. *Static Detection of Dynamic Memory Errors*. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96). Philadelphia, PA. p. 44-53. May 1996.

[Evans99a] David Evans and Andrew Twyman. *Policy-Directed Code Safety*. Proceedings of the 1999 IEEE Symposium on Security and Privacy. Oakland, California. p. 32-39. May 1999.

[Evans99b] David Evans. *Policy-Directed Code Safety*. MIT PhD Thesis. February 2000.

[Evans2000a] David Evans. *Annotation-Assisted Lightweight Static Checking*. In *The First International Workshop on Automated Program Analysis, Testing and Verification*. February 2000.

[Evans2000b] David Evans. *Let's Stop Beating Dead Horses and Start Beating Trojan Horses*. Short presentation at the *Infosec Research Council, Malicious Code Study Group*. San Antonio, Texas. January 2000.

[Fournet96] Cédric Fournet and Georges Gonthier and Jean-Jacques Lévy and Luc Maranget and Didier Rémy. *A calculus of mobile agents*. Lecture Notes in Computer Science, Springer-Verlag. Volume 1119. 1996

[Freeman99] Eric Freeman, Susanne Hupfer and Ken Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison Wesley, 1999.

[Gelernter85] David Gelernter. *Generative communication in Linda*. ACM Transactions on Programing Languages and Systems. Volume 7, Number 1, p. 80-112. January 1985.

[IEEE2000] *IEEE Std 802.11-1997: Short description of the standard.* http://www.manta.ieee.org/groups/802/11/main.html

[Intanagonwiwat2000] Chalermek Intanagonwiwat, Ramesh Govindan and Deborah Estrin. *Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks.* To appear in *ACM MobiCom 2000*

[Kahn99] J. M. Kahn, R. H. Katz and K. S. J. Pister. *Next Century Challenges: Mobile Networking for Smart Dust*. ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 99). p. 217-278. August 1999.

[Lehman98] Tobin J. Lehman, Stephen W. McLaughry and Peter Wyckoff. *T Spaces: The Next Wave*. IBM Systems Journal, August 1998.

[Long99] T. J. Long, B. W. Weide, P. Bucci, and M. Sitaraman. *Client View First: An Exodus From Implementation-Biased Teaching*. Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education. p. 136-140. March 1999

[Mockapetris87] Paul V. Mockapetris. *Domain Names – Concepts and Facilities*. Network Working Group, RFC 1034. November 1987.

[Orcero2000] David Santo Orcero. *The Code Analyser LCLint*. In *Linux Journal*, May 2000.

[Picco99] Gian Pietro Picco, Amy L. Murphy and Gruia-Catalin Roman. *LIME: Linda Meets Mobility*. Proceedings of the 1999 International Conference on Software Engineering. p. 368-377. May 1999.

[Pramode2000] Pramode C E and Gopakumar C E. *Static checking of C programs with LCLint*. In *Linux Gazette*, March 2000.

[Roman97] Gruia-Catalin Roman, Peter J. McCann and Jerome Y. Plum. *Mobile UNITY: reasoning and specification in mobile computing.* ACM Transactions on Software Engineering and Methodology. Volume 6, Issue 3. p. 250-282. July 1997.

[Rotman2000] David Rotman. *Molecular Computing*. Technology Review. May/June 2000.

[Subramanian2000] Lakshminarayanan Subramanian and Randy H.Katz. An Architecture for Building Self-Configurable Systems. To appear in IEEE/ACM Workshop on Mobile Ad Hoc Networking and Computing. August 2000.

[Tennenhouse2000] David Tennenhouse. *Embedding the Internet: Proactive Computing*. Communications of the ACM. Volume 43, Issue 5. p. 43-50. May 2000.

[Twyman99] Andrew R. Twyman. *Flexible Code Safety for Win32*. MIT SM Thesis, May 1999.

[Viega2000] John Viega and David Evans. *Separation of Concerns for Security*. Workshop on Multi-Dimensional Separation of Concerns in Software Engineering. p. 126-129. February 2000.

[Waldbauer2000] Gilbert Waldbauer. *Millions of Monarchs, Bunches of Beetles: How Bugs Find Strength in Numbers*. Harvard University Press. 2000.

[Weide99] Bruce Weide. *Software Component Engineering*. Ohio State University, 1999. http://www.cis.ohio-state.edu/~weide/sce/now/index.html.