Orchestrating Concurrency in Robot Swarms

Anthony Cowley and C.J. Taylor
GRASP Lab
University of Pennsylvania
Philadelphia, PA 19104-6228
Email: {acowley, cjtaylor}@seas.upenn.edu

***Abstract*— A functional approach to programming robot swarms brings with it well-defined properties that allow for automated concurrency and distributed execution. Further, the particular expressiveness of a pure functional language with first-class closures captures so cleanly certain biologically-inspired behaviors that program specification often becomes compact enough to allow a programmer to visually inspect the program code for the entire swarm. This benefit comes in contrast to more piece-meal construction methods used to build-up robot software from discrete components. While such programming models capture the engineered structure of a robotic system, the dynamic, decentralized qualities sought after in robot swarms are well-matched by the idioms of functional concurrent programming.**

I. Introduction

Programming a swarm of robots suggests a number of features found rarely in traditional software applications. Concurrency – the simultaneous execution of multiple program elements – is a fundamental feature of swarms, and can not be viewed merely as a performance boost. Although concurrency is, today, receiving a great deal of attention thanks to the shift to multi-core processors, swarms of robots provide a computational platform that requires software designed to operate over tens or hundreds of logical processors, rather than the two or four cores found in typical consumer products. Such levels of concurrent execution suggest that manual annotation of critical sections – sequences of program instructions that only one processor may be executing at a time – may be a crippling limitation. Instead, swarms show their strength when executing programs capable of scaling to accommodate, and take advantage of, an arbitrary number of nodes.

Swarms terminology, together with the associated biomimetic metaphors, further evokes notions of fault tolerance commonly seen in nature, yet seldom found in man-made systems. In fact, applying a simplistic view of programming, where a program is contained in a plain text file, to any system implies the immediate imposition of a single point of failure: the interpreter, or runtime, executing the program. Yet the single-file view of a program is nonetheless appealing because it suggests a self-contained specification for system behavior. Ideally, swarms programming should be as comprehensible to the programmer as a traditional program entirely contained in a single file, while as fault tolerant as a system composed of loosely coupled, often redundant, components. In essence, the programmer should be specifying something akin to DNA that may be executed, spliced, and spread throughout the swarm in a completely dynamic manner.

A. *Functional Swarms*

Functional programming, a branch of computer science loosely identified with the representation of program semantics solely in terms of idempotent function calls, is enjoying a resurgence due to the nature of progress in computing hardware. Rather than faster CPUs, today's systems gain performance by adding logical processors. While this trend has only recently achieved significant penetration of the consumer electronics market segment in the form of multi-core processors, it is firmly entrenched in the data center, where companies such as Google are harnessing thousands of discrete processors to solve computations involving tremendous amounts of data. In order to take advantage of all these processors, and the memory physically attached to them, Google has employed MapReduce [1] as a way of specifying an operation to be carried out over a large dataset. The functional specification of the operation eases the distribution of the program over the many available processors without specifically addressing concurrency or distribution.

Robotics programming has also seen applications of functional programming in the areas of specifying reactive behaviors in the form of pure functions of sensor inputs tightly coupled to evince desired behaviors [2] [3]. But swarms of robots represent a computational platform that has as much in common with one of Google's data centers as it does the collection of sensors and actuators that make up a single robot. Indeed, many swarm behaviors are well-captured by a recursive behavior definition that may be simultaneously applied over any number of robots. Additionally, proper encapsulation of recursion semantics allows a behavior to recursively apply itself to other agents. Such a capability enables the spread of live behaviors throughout a swarm, and for the sharing of composable behavioral components learned during execution. Finally, this specification style leads to compact encodings of behaviors – short programs! – that, despite their single point of definition, are fully capable of expressing highly decentralized behaviors that are robust to individual failures among the swarm population.

B. Related Work

The dominant manner of creating software for robotic systems is to reverse the traditional software engineering methodology of top-down specification, implementation, refinement iterations, and take a bottom-up approach by creating small components in a dataflow programming style as in Player [4], ROCI [5], and many other robotics programming frameworks. In this design paradigm, one builds up a high-level application by creating a graph with, preferably, directed edges between functional components. Such a technique alleviates much of the pain of concurrent software design, while making distributed execution essentially transparent to the component developer. These benefits may be viewed as a result of designing a large system as a composition of many smaller, functionally independent parts whose successful concurrent execution is achieved by building upon message passing to define control flow [6] (rather than the synchronous procedure calls that define control flow in most mainstream, imperative languages such as Java, C, C#, etc.), or they may be viewed as a result of a specialized runtime subsuming the top-levels of a traditional program. In any case, the key insight of such a design is to elevate the data a program operates on to a position of greater prominence than it typically has in an imperative design, in which control flow receives the most attention. To wit, an imperative program is a carefully chosen sequence of procedure calls, while a dataflow graph is a declaration of what data goes where.

Breaking a problem down into sub-problems is sound engineering, and is a natural way to build a robot. Strict adherence to this methodology leads to an intuitive mapping between software and hardware components, so it is no great stretch to compose the software that drives a robot from components that either interface with hardware, or provide a specific abstraction or computation. Connecting the resultant components leads to the ascendancy of data in the dataflow programming paradigm common to robotic software development today. Yet the correlation between hardware and software components can also be a limiting metaphor if it ignores the potential fluid qualities of software. The act of laying out a circuit board, or connecting software components, may yield a fine machine, but typically lacks the expressiveness to capture a dynamic exchange of functionality between units. Indeed, there is little evidence of such behavior in most mechanical systems. However, this very capability can be used to effectively specify inter-agent coordination, and enable the exchange of functionality among members of the swarm.

II. A Scheme for Swarms

The prominent role of data in component-based programming paradigms provides for the flexible topology specification of dataflow models, but it generally stops short of representing *all* system specifications (*i.e.* traditional inert data, function parameters, and executable code) as exchangeable data. Such a duality of data and program code is a very familiar concept to Lisp [7] programmers: Lisp (LISt Processor) programs are represented as lists in the Lisp programming language. This co-definition of code and data allows for great flexibility in self-modifying programs, but it also speaks to an ideal that is further reinforced by the *continuation* data structure, as found in the Scheme [8] programming language (Scheme is a descendant of Lisp), which is itself a bidirectional transformation between the state of execution of a program and a data structure. That is, any given point in a program's execution may be stored as a data structure to be passed to other functions, and invoked any number of times. This unification of the data a program operates over, the program code itself, and any particular state of execution as data structures captured in the programming language shatters the limitation of a simple program implying a simple, centralized execution model. Furthermore, an adherence to the lambda calculus [9], with no allowance for mutable state, and faithfulness to Scheme's lexical scoping rules, provides for a clean, concise definition of how the various expressions that make up a program may be executed. There are no questions of shared state, and there is no question of side-effects (*e.g.* a sub-procedure that changes a variable that the calling procedure refers to). The semantics of function calls are distilled to their bare minimum, so that an interpreter can safely, with respect to internal integrity, execute many function calls concurrently, and easily move, or replicate, control flow around the swarm.

Traditionally, and in Scheme, a *lambda* [10] [11] expression captures the environment in which it is evaluated, and defines a closure. The resulting structure may be thought of as a function of some number of parameters in the standard mathematical sense, or it may be thought of as a suspension since the body of the lambda expression is not evaluated. Instead, the environment in which the lambda expression was encountered is stored, along with a "slot" for any formal parameter of the function. The resulting data structure is a closure, which may be evaluated at a later time by binding values to the function parameters (filling the empty slots), and executing any sub-expressions specified in the body of the lambda expression. When the binding and execution (*i.e.* invocation) occurs, the previously suspended evaluation of the lambda expression is resumed: the closure's environment is extended by the binding of any formal parameters to values passed to the invocation, and the function's body is then evaluated. Scheme treats functions defined in this way as *first-class* entities. This means that they are stored as data structures that may be passed to, or returned from, other functions.

III. Programming the Swarm

The priorities when designing software for a swarm of robots are concurrent execution, distributed control, fault tolerance of high-level behaviors, and comprehensibility for the programmer. Any effort to automate decisions regarding potential concurrent evaluation of elements of a single program is greatly aided by the lack of shared state implied by the immutability of state in a pure functional language. A system for the distribution of control flow is bolstered by

first-class continuations. Fault tolerance is achieved on the macro-scale by the distribution of control flow: centralized behaviors on a small scale may fail without affecting the continued execution of self-contained program fragments that have been distributed across the swarm. Finally, the programmer should ideally be able to digest the sum-total of programming for a swarm by inspecting a single source file of manageable length.

A. System Implementation

System development began with a working foundation for single-robot software development. The ROCI platform supports the creation of dataflow-style applications, and can interface with many platforms via full support for HTTP access to all internal data [12]. That foundation includes a declarative scripting system that involves creating XML files for specifying component connection topologies for various behaviors. In this manner, high-level behaviors, such as waypoint navigation, may be aliased to a script (*e.g.* "go_to"), which specifies how a GPS interface should talk to a navigation controller, which should talk to an obstacle avoidance controller, which pulls data from a range sensor, etc. In effect, there is a "go_to" machine that is captured by the script file. This machine is parameterized by the destination waypoint, which is fed to a particular input as specified in the script. For pedagogical purposes, that scripting system was interfaced to the Ruby programming language in order to present students with a robot programming system that had minimal syntax, yet could cleanly interact with the wealth of components written in lower-level languages to interface with hardware devices and implement conceptually complex computations.

With this background, an interpreter for a subset of the Scheme programming language was written in Ruby. The dialect of Scheme supported by the intepreter has built-in support for automated concurrent evaluation of parallelizable statements (*e.g.* functions applied to a list with the higher-order *map* function, or expressions in the body of a provided *scatter* function that evaluates all expressions in its body concurrently). The interpreter will also pass a continuation to another node if that node is a formal parameter of the continuation, and the current host is not. This distribution scheme is not always optimal, but it captures the intended behavior in many cases, without the programmer having to consider where best to evaluate a given expression, or whether multiple expressions can be evaluated concurrently. The strict scoping rules of Scheme, along with immutable data, mean that the system's efforts to distribute, or otherwise concurrently execute, program statements are always safe in terms of program specification.

The primary focus of the Scheme interpreter is to provide lexical closures and first-class continuations in order to make concurrency safer, and automatic distribution possible. The interpreter provides tail-call optimizations to allow for a highly-recursive programming model, and is capable of serializing any closure to a JavaScript Object Notation (JSON) object that may be transmitted to a network peer. The JSON model is attractive since its syntax matches the contents of a typical closure. That is, JSON is specifically designed for representing associative lists (such as the binding table in a closure) as well as anonymous lists (used throughout Scheme programming).

Fragment 1 Allocation With Constraints

```
1: (alloc 1 (lambda (r) (== (r.color) "green")))
2: (lambda (leader)
3:     (alloc 3 (lambda (r) (< 5 (dist r leader))))
```

B. Functional Composition

Behavior composition is readily achieved with the use of higher-order functions. A simple usage of the lambda syntax is seen in the *alloc* function that finds free robots. The *alloc* syntax is actually a point of multiple dispatch for two allocators: one that simply takes the number of free robots to find, and another that takes the number of free robots to find along with a function that must evaluate to *true* when given a candidate robot. This can be seen in Line 1 of Fragment 1, where only green robots are desired. The constraint function may be a composition of any number of other functions joined in any way. The only restriction is that the composition must take a single parameter (the robot to test) and return *true* or *false*. While this simple example demonstrates in-line function creation, it is also important to remember that lambda expressions capture their execution context. Thus, the function passed to *alloc* may refer to any variables in the scope in which it is defined, as shown in Lines 2-3 of Fragment 1. This constraint function will execute on each free robot the system can find, and that execution will correctly refer to the robot bound to the *leader* identifier.

C. Clearing a Minefield

Higher-order functions and continuation passing enable the concise specification of many highly parallel activities. Consider the example of clearing a minefield. There are a number of features that are needed to have an easily expressed, scalable solution: viral spread of behavior, constraint composition, and concurrent, distributed execution. The mine clearing example excerpted in Fragment 2 showcases these features.

The mine-clearing program, shown in its entirety in Fragment 2, begins with the concurrent application of the *search* function to a number of robots capable of detecting mines on Line 24 (red robots can detect mines in this simulation). This expression utilizes the *map* higher-order function that, in this interpreter, concurrently applies the function supplied in the first parameter to each element of the list supplied as the second parameter. The *search* function continuously (via self-recursion) navigates the mine-seeking robot around the field until a mine is detected. At this point, a new function is called. This function utilizes a constrained allocation in Line 8 to find a free robot capable of digging (green robots are capable of digging up mines). A function that guides

Fragment 2 Mine Clearing

```
1:  (define neutralize
2:      (lambda (seeker mine)
3:          (seeker.go_to (seeker.pos_x) (seeker.pos_y))
4:          ((lambda (digger)
5:              (digger.go_to (car mine) (cdr mine))
6:              (digger.wait)
7:              (digger.dig))
8:           (alloc 1 (lambda (r) (== (r.color) ”green”))))))
9:
10: (define search
11:     (lambda (r)
12:         (r.go_to (- (* (rand) 12) 6) (- (* (rand) 12) 6) 4)
13:         (letrec ((check
14:                     (lambda ()
15:                         (letrec ((mine (r.detect_mine)))
16:                             (if (not (null? mine))
17:                                 (neutralize r mine)
18:                                 (if (r.busy)
19:                                     (check)
20:                                     #t))))))
21:             (check))
22:         (search r)))
23:
24: (map search (alloc 5 (lambda (r) (== (r.color) ”red”))))
```

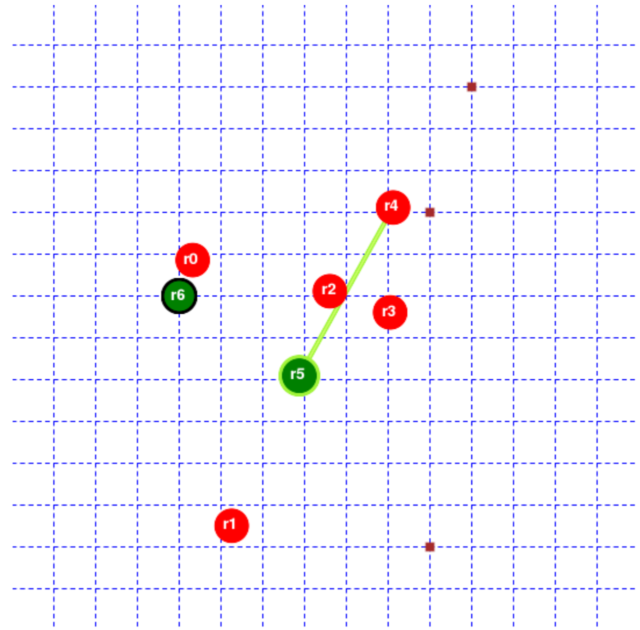

Fig. 1. Digging robots, r6 and r5, are recruited by mine-seeking robots to help clear a minefield (mines are represented by small squares). Here, r6 is helping r0 clear a mine, while r4 is transmitting a closure to r5 for evaluation. The execution of this closure will drive r5 to the mine r4 has detected, and initiate the digging sequence.

the digger robot to the mine, and digs up the mine, is then sent to the digging robot simply by invoking that function with the digger robot as an argument. This behavior is shown in Fig. 1, in which the passing of a closure from a seeker robot to a digger robot is visualized by a line connecting two robots. This example demonstrates a specific centralized behavior requiring inter-agent synchronization – the act of cooperatively digging up the mine – within the parallel execution of the larger task of clearing the mine field. The available flexibility in behavior specification allows the programmer to mix-in centralized behaviors on a small-scale without sacrificing the robustness of large-scale swarm capabilities. In this case, the spread of behavior is finite: once the mine is cleared, the digger robot becomes free again so that it may assist another seeker robot. However, a spreading behavior need not terminate so quickly; any behavior can replicate itself indefinitely.

D. Viral Behavior

A more dramatic viral spread of behavior is illustrated by the evolution of a viral behavior released into a swarm, as shown in Fig. 2. A simple function is initially applied to a single robot, but the behavior “infects” any free robots that come into contact (*i.e.* become reachable on a network, here determined by distance) with a robot executing the viral function. Such an ability is highly valuable in real-world deployment scenarios, where many robots may not always be reachable on the network. In this example, a single robot is given a behavior with decaying “infectiousness” that causes the infected robot to wander around until it has come into network range of, and infected, some number of robots before turning into a star. The members of the swarm start out at random locations unknown to any other robot, and are not displayed in the simulation until they have been

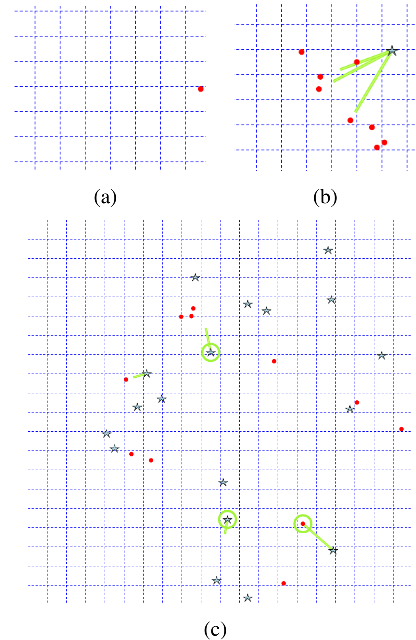

Fig. 2. Three stages of a viral behavior. The behavior begins with a single agent (a), which spreads an infectious behavior among connected peers that terminates with the agent visualized by a star shape (b). Eventually, a large number of agents have been recruited without any centralized organization (c).

discovered by an infected robot. In this manner, a behavior is spread throughout an initially disconnected swarm without any centralized allocation capability. This example controls total infection by decrementing a counter passed with each recursive invocation of the behavior, and demonstrates the

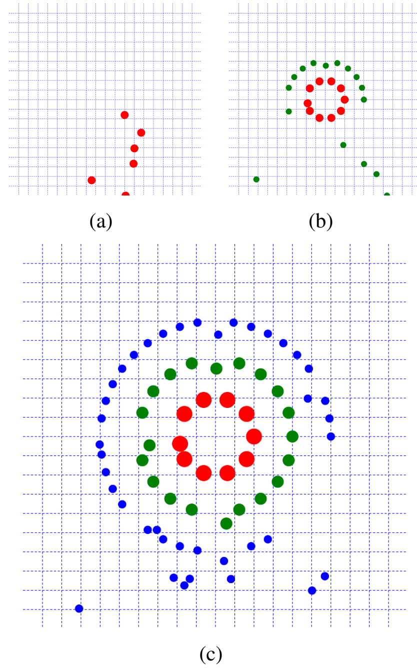

Fig. 3. Distributed assembly, pattern generation, and formation control can be implemented in discrete phases. One example of the utility of such an approach is that closed perimeters may be established in sequence to avoid the problem of agents getting trapped inside structures as they form.

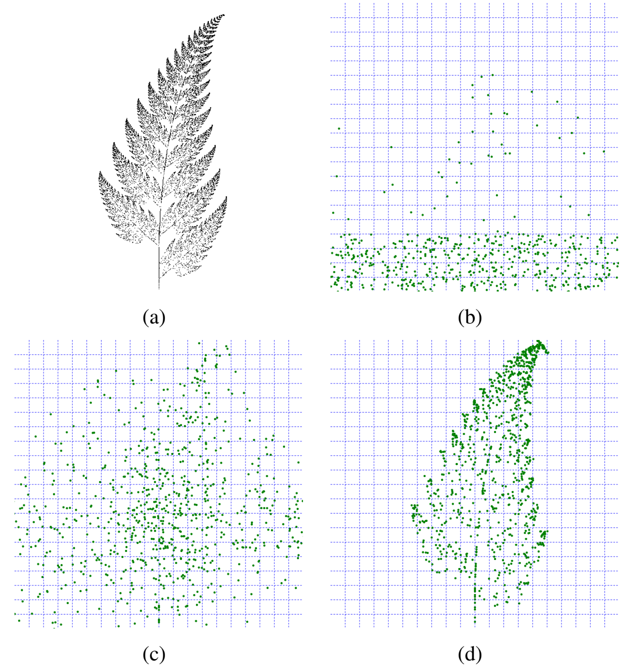

Fig. 4. (a) *Barnsley's Fern* as shown by running a probabilistic iterated function system 30,000 times. (b-d) Stages in the construction of the same fern by 1,000 kinematically simulated robots independently evaluating the same function system.

ability to specify an approximate desired recruitment pool size to be drawn from an unknown population.

E. Multi-phase Assembly

The pattern generation work of Hsieh and Kumar [13] provides a clean example of provably convergent, decentralized control achieving global geometric goals (*i.e.* distributing robots along a perimeter) while maintaining local constraints such as communication signal strength. Utilizing this style of controller as a low-level capability of members of the swarm, assembly tasks may be structured to provide the necessary sequentiality to ensure successful construction of more complex geometries. Fig. 3 illustrates three stages in the construction of a series of concentric circles. By sequencing the construction of closed geometries, one can guarantee that inner areas are accessible when necessary, and that components of the outer structures not be imprisoned by the closing of an inner structure. The structuring of this program is completely captured by sequential applications of the *map* function, each of which concurrently assembles one section of the structure.

F. Organic Structures

This structuring capability provides a framework upon which to build more decentralized assembly tasks, such as organic growth [14]. In nature, many similarly-capable elements, from cells to termites, come together to build astonishingly complex structures. Establishing simple rules that, when executed across a multitude of agents, cause seemingly high-level organization to emerge is a goal of much swarms research. Such rules may most easily be leveraged if their application can be deliberately structured in some manner, or applied chaotically within a deliberatively structured framework. Furthermore, the ability to freely exchange these rules and behaviors among agents can enable the emergence of isolated hierarchies at various scales, thereby allowing for a full range of expression in assembly tasks.

Consider the iterated function system [15] that yields Barnsley's Fern, as shown in Fig. 4(a). This system can be implemented recursively in Scheme to provide a controller that may be simultaneously evaluated by 1000 robots as in Fig. 4(b-d). While such a controller allows for a maximal amount of concurrency, and results in a pleasingly self-similar structure highly reminiscent of the Black Spleenwort fern, it is impractical to have the structure's growth dynamically respond to external stimuli due to the complete lack of run-time coordination among members of the swarm. In other words, the ultimate organization demonstrated by this controller is solely a result of the initial coordination of sharing a particular set of functions. However, it can be highly useful for the swarm to adjust its behavior in order to respond to the environment or unpredicted operating conditions. For example, a natural application of a plant-like structure is as a scalable method of deploying solar collectors. Yet, while natural plants are perfectly capable of adjusting their growth to aim towards areas of greater sunlight, the iterated function system must be strictly coordinated among all agents to obtain an organized result. That coordination largely precludes reactive behaviors triggered by individual agents. Instead, one may implement a controller capable of adjusting growth parameters in response to external stimuli

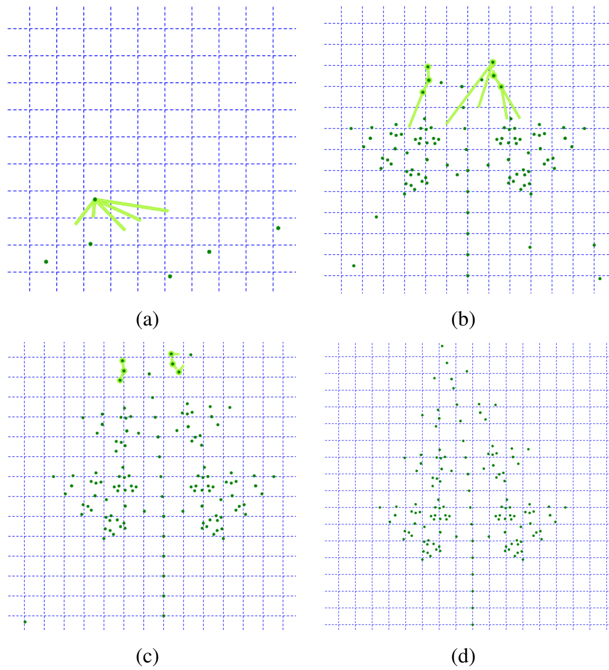

Fig. 5. A recursively-constructed fern-like structure grown among 165 robots. The simple program that generates this structure is given to a single robot which recursively invokes the same structuring behavior on other robots (distribution of control flow is indicated by an animated line connecting two robots).

by building the structure in a manner closer to that used by nature: hierarchically. Such a controller, implemented in a few tens of lines of code, injected into a single "root" robot gives rise to the recursively constructed form shown in Fig. 5. The growth of this structure is a dynamic process capable of responding to stimuli sensed by individual robots.

G. Future Work

While a pure functional language with no mutable state enables aggressive concurrency in many circumstances, it does not eliminate the fact that the swarm, as it exists in the real world, does have state. For example, the swarm has a finite number of members, which means that there may be resource contention issues when trying to find free robots. While mechanisms based on orderings and timeouts may be applied to alleviate some types of contention, the fact remains that this issue is difficult to consider for the swarm programmer. One could argue that resource contention should not be the concern of the programmer, that the system should handle it, but the dynamic nature of available resources in a sometimes-connected swarm setting means that a failure to obtain required resources should be taken as a likely circumstance that, if encountered, does not represent a permanent setback. To combat this, behaviors can be designed in an incrementally expanding fashion, rather than an all-or-nothing proposition. A robot can try to spread a behavior virally, rather than requiring an instantaneous allocation. Furthermore, a behavior can be suspended until some necessary conditions are met. This powerful approach may be captured by the threading of a continuation through another, secondary behavior. A suspended behavior may, at any time, be resumed right where it left off by an invocation of the continuation. This enables individuals to continue executing one behavior while periodically checking if current conditions allow some other behavior to continue. The exploration of this style of behavior specification is left to future works.

IV. Conclusion

The system presented here extends the utility of existing software components. Such components provide a rich vocabulary from which to specify the software machinery necessary to perform a wide variety of behaviors, yet lack the expressiveness of dynamism that software is capable of. This dynamism, realized in the fluid exchange of functionality, represents a crucial capability when attempting to orchestrate the activities of a swarm of robots.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Datat Process on Large Clusters," in *OSDI'04, 6th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004, pp. 137–150.

[2] J. Peterson, P. Hudak, and C. Elliott, "Lambda in Motion: Controlling Robots with Haskell," *Lecture Notes in Computer Science*, vol. 1551, pp. 91–105, 1999.

[3] J. Peterson and G. Hager, "Monadic Robotics," in *Domain-Specific Languages*, 1999, pp. 95–108.

[4] B. Gerkey, R. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. Mataric, "Most Valuable Player: A Robot Device Server for Distributed Control," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2001, pp. 1226–1231.

[5] A. Cowley, L. Chaimowicz, and C. J. Taylor, "Design Minimalism in Robotics Programming," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 31–37, March 2006.

[6] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Journal of Artificial Intelligence*, June 1977.

[7] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, April 1960.

[8] G. J. Sussman and G. L. Steele, Jr., "Scheme: An Interpreter for Extended Lambda Calculus," MIT AI Lab, AI Lab Memo AIM-349, December 1975.

[9] A. Church, "An Unsolvable Problem of Elementary Number Theory," *American Journal of Mathematics*, vol. 58, pp. 354–363, 1936.

[10] G. L. Steele, Jr. and G. J. Sussman, "Lambda: The Ultimate Imperative," MIT AI Lab, AI Lab Memo AIM-353, March 1976.

[11] ——, "Lambda: The Ultimate Declarative," MIT AI Lab, AI Lab Memo 379, November 1976.

[12] A. Cowley, H. Hsu, and C. J. Taylor, "Opening the Dialog," in *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, 2006, pp. 2775–2781.

[13] M. A. Hsieh and V. Kumar, "Pattern Generation with Multiple Robots," in *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, 2006, pp. 2442–2447.

[14] R. Nagpal, "Self-Organizing Shape and Pattern: From Cells to Robots," *IEEE Intelligent Systems*, vol. 21, no. 2, pp. 50–53, March/April 2006.

[15] M. F. Barnsley and S. Demko, "Iterated Function Systems and the Global Construction of Fractals," in *The Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 399, no. 1817, 1985, pp. 243–275.