Swarm-Oriented Programming of Distributed Robot Networks

Carlo Pinciroli and Giovanni Beltrame

June 20, 2016

Abstract

From delivery drones to self-driving cars, robots are becoming increasingly pervasive and interconnected in our society: soon, programming large, distributed robotic teams will be the norm. One of the current challenges is the definition of suitable programming primitives that generate well-structured, reusable, and predictable team behaviors. We present a language construct, called `swarm`, that overcomes the all-or-nothing choice between focusing either on individual robots or the team as a whole. Our idea is a meet-in-the-middle approach: The `swarm` construct allows the developer to categorize the robots with respect to certain conditions, and to assign tasks to the robots that belong to a certain `swarm`. Through code examples, we show how the programs produced through this construct are structured into well-defined, short, and reusable functions.

Introduction

The incoming robotics revolution will fundamentally change the type of devices available to the general public. In contrast to the current concept of ‘smart’ devices, which are capable of sensing and computation, robots are devices that integrate sensing, computation, and actuation in the physical world. An inescapable consequence of this introduction of robotics into our daily lives is the fact that these devices will be networked, will be heterogeneous in terms of capabilities, and will need to coordinate in teams to achieve complex tasks.

The problem of coordinating mid- and large-scale teams of robots (*swarms*) is receiving increasing attention in the research community [1]. Current approaches range from centralized methods, in which a single planner collects information and dictates the actions of the robots, to decentralized methods, in which every robot acts accordingly to the limited information available on-board. Decentralized methods promise better robustness to failure and better parallelism with respect to centralized methods, at the cost of more complex design.

The complexity of designing decentralized robotics systems hinges around the fact that functional requirements are expressed at the swarm level, while actions must be executed at the individual level. This is often referred to as ‘global-to-local problem’. During development, this issue turns into the problem of selecting a suitable level of abstraction to express swarm behaviors comfortably. The current approaches to swarm programming can be categorized by the granularity of their level of abstraction. Broadly speaking, two families of approaches are commonly followed: the bottom-up approach, and the top-down approach.

In the bottom-up approach, the developer focuses on the individual behaviors. The development process proceeds by trial-and-error through progressive refinement of individual behaviors, until the requirements at the swarm level are satisfied. Programming languages such as C, C++, and Python, often integrated with frameworks such as Robot Operating System (ROS, `http://www.ros.org`) [2] or DroneKit (`http://www.dronekit.io`), are common choices to pursue this approach. At the opposite side of the spectrum is top-down development, in which the focus is on the swarm as a whole. A number of domain-specific languages belong to this category, such as Proto/Protelis [3, 4] (for aggregate programming) and Meld [5] (for self-assembling robots).

These two families of approaches show benefits and drawbacks. Bottom-up development offers a high degree of control on the system; however, it exposes the developer to an overwhelming amount of detail, including communication protocols, low-level control, and (with C/C++) even memory management. Top-down development, on the other hand, allows the developer to concentrate on purely swarm-related algorithms, sacrificing design flexibility on many aspects of the system, such as actuation.

In this article, we argue that the current all-or-nothing choice between bottom-up and top-down programming can be overcome. We present a language construct, called `swarm`, that allows one to tag robots that respect specific conditions dictated by the developer, and to assign tasks to robots as a function of their tags. Robots can be given multiple tags, and conditions for task assignment can be complex, e.g., require a robot to have multiple tags. We believe that the ability to tag robots conditionally offers a mixed level of abstraction, that enables the developer to express complex swarm algorithms in a comfortable and concise manner. The `swarm` construct is one of the novel aspects of the recently introduced Buzz programming language [6] (`http://the.swarming.buzz`).

The Buzz Programming Language

Buzz is a programming language for swarm robotics [6]. The main driver in the design of Buzz is to expose to the developer a small, but powerful set of primitives that enables the expression of complex swarm algorithms.

Buzz is also designed as an extension language. Rather than replacing the software stack present on a robot, Buzz is conceived to integrate with it seamlessly, exposing only the relevant aspects of a robot API. This design choice makes it possible to use Buzz with virtually any type of robot, and to add application-specific primitives to the language. In a heterogeneous robot swarm in which different software frameworks are employed (e.g., ROS for ground-based robots and DroneKit for quad-rotors), a Buzz layer effectively eliminates the low-level differences and enables code reuse and performance comparison.

From the computational point of view, Buzz sees a robot swarm as a discrete collection of interacting robots. Each robot runs an instance of the Buzz Virtual Machine (BVM), on which a Buzz script is executed. In the current version of the Buzz run-time, it is assumed that the same script runs on every robot.

The robots can be heterogeneous in their capabilities, and must be able to exchange information and detect each other. In particular, Buzz is based on a form of inter-robot communication called *situated communication*. This communication modality is based on a local message broadcast which is also ‘situated’ because a robot, upon receiving a message, is capable of detecting the position of the sender with respect to its own frame of reference. Situated communication forms the base of a large number of swarm algorithms, including pattern formation, task allocation, aggregation, exploration, and many more.

At first sight, the Buzz syntax resembles JavaScript, Python, and Lua. This choice was made to ensure a short learning curve for newcomers. The primitives offered by Buzz include classical constructs, such as variable and function definitions, branches, and loops; and more high-level constructs designed for spatial and network coordination. The `neighbors` construct, for instance, is a data structure that contains information about the robots in communication range with a certain robot. Through this construct, it is possible to broadcast messages, aggregate local information, filter robots and their associated information, and spatially interact with nearby robots. To propagate information globally across the swarm, Buzz offers the `stigmergy` construct [7], a light-weight distributed hash table based on local broadcast and resistant to severe message loss.

The `swarm` construct

The `swarm` construct of Buzz allows the programmer to tag a set of robot according to a certain condition. While the `swarm` syntax is simple, the range of possible applications is wide, making the `swarm` construct a powerful programming tool.

Programming Interface

Creation and destruction. The creation of a swarm occurs through a call to the `swarm.create()` method. This method takes a unique numerical identifier as parameter, which is internally used by the BVM to manage swarm membership.

```
s = swarm.create(1)
```

Once a swarm has been created, it is empty. To assign robots to it, two methods are available: `swarm.select()` and `swarm.join()`. The first method allows the developer to express a condition evaluated individually by every robot at run-time. If the given condition is satisfied, the robot joins the swarm. The `swarm.join()` method lets a robot join a swarm unconditionally.

```
# If the robot identifier is even, the robot joins the swarm
s.select(id % 2 == 0)
# Every robot joins the swarm unconditionally
s.join()
```

The logic of leaving a swarm is analogous, and achieved through the methods `swarm.unselect()` and `swarm.leave()`.

Swarm lambdas. A *swarm lambda* is a function assigned to a specific swarm for execution. When a swarm lambda is specified, every robot in the swarm executes it:

```
# All the drones in swarm s take off
s.exec(function() { takeoff() })
```

Internally, the lambda is executed as a *swarm function call*. This call modality differs from classical function calls in that the current swarm id is pushed onto a dedicated *swarm stack*. Upon return from a swarm function call, the swarm stack is popped. The swarm stack is managed by the BVM to keep track of nested swarm function calls. When the swarm stack is non-empty, the `swarm.id()` method is defined. If called without arguments, it returns the swarm id at the top of the swarm stack (i.e., the current swarm id); if passed an integer argument `n` > 0, it returns the `n`-th element in the swarm stack. The swarm stack is used by other Buzz primitives, such as `neighbors` and `stigmergy`, to limit the scope of their operations to the robots in the stack-top swarm.

Set operations. Swarms can be considered sets of robots. Therefore, it is possible to apply to them standard set operations, such as intersection, union, difference, and negation. Each of these operations results in the creation of a new swarm.

```
# We assume swarms s1 and s2 have been created and filled
# Intersection of s1 and s2 (robots belonging to both s1 and s2)
# Parameters: the swarm id, and the list of swarms to intersect
i = swarm.intersection(100, s1, s2)

# Union of s1 and s2 (robots belonging to s1 and/or s2)
# Parameters: the swarm id, and the list of swarms to merge
u = swarm.union(101, s1, s2)

# Difference of s1 and s2 (robots belonging to s1 and not s2)
# Parameters: the swarm id, and the list of swarms to consider
d = swarm.difference(102, s1, s2)

# Negation of s1 (robots not belonging to s1)
n = swarm.others(103)
```

Swarm membership dynamics. Through the combined use of `swarm.select()`, `swarm.unselect()`, and the set operations, it is possible to express complex dynamics that evolve over time. The `swarm.select()` method is useful to create swarms according to conditions that are satisfied at the specific time they are evaluated. It is important to notice that the condition that defines a swarm created through this method might become false at some point, without the swarm composition being affected. If the condition is intended as a membership requirement, however, the `swarm.unselect()` method offers a simple way to enforce this requirement. In contrast, set operations can be used to manipulate swarms according to their current composition, rather than on the condition they satisfied at any moment in time. In other words, set operations make it possible to select robots that are (or not) part of one or more swarms, effectively treating swarm membership as a sort of 'tag' associated to a robot. As it will be shown in the following, a careful definition of the swarms allows one to structure a complex swarm algorithm into a well-defined set of small, highly reusable functions.

Low-level Implementation

The management of swarm information is performed transparently by the BVM. Essentially, each robot maintains two data structures about swarms: the first structure concerns the swarms of which the robot is a member; the second stores data regarding neighboring robots.

Membership management. Every time a `swarm.create()` method is executed, the BVM stores the identifier of the created swarm into a dedicated hash table, along with a flag encoding whether the robot is a member of the swarm (`1`) or not (`0`). Upon joining a swarm, the BVM sets the flag corresponding to the swarm to `1` and queues a message `<SWARM_JOIN, robot_id, swarm_id>`. Analogously, when a robot leaves a swarm, the BVM sets the corresponding flag to `0` and queues a message `<SWARM_LEAVE, robot_id, swarm_id>`. Because leaving and joining swarms is not a particularly frequent operation, and motion constantly changes the neighborhood of a robot, it is likely for a robot to encounter a neighbor for which no information is available. To maintain everybody's information up-to-date, the BVM periodically queues a message `<SWARM_LIST, robot_id, swarm_id_list>` containing the complete list of swarms to which the robot belongs. The frequency of this message is chosen by the developer when configuring the BVM installed on a robot.

Neighbor swarm data. The BVM stores the information in a hash map indexed by robot id. Each element of the hash map is a `(swarm_id_list, age)` pair where `swarm_id_list` corresponds the list of swarms of which the robot is a member, and `age` is a counter of the time steps since the last reception of a swarm update. Upon receipt of a swarm-related message (i.e., `SWARM_JOIN`, `SWARM_LEAVE`, `SWARM_LIST`), the BVM updates the information on a robot accordingly and zeroes the `age` of the entry. This counter is employed to forget information on robots from which no message has been received in a predefined period. When the counter exceeds a threshold decided when configuring the BVM, the information on the corresponding robot is removed from the structure. This simple mechanism allows the robots to commit memory storage on active, nearby robots while avoiding waste of resources on unnecessary robots, such as out-of-range or damaged robots. In addition, this mechanism prevents excessive memory usage when the swarm size increases.

Message queue optimizations. To minimize the bandwidth required for swarm management, the BVM performs a number of optimizations on the queue of swarm-related outbound messages. The optimizations involve identifying the minimal number of messages to send in order to communicate the current state of swarm membership of a robot. The details on this aspects exceed the scope of this article, and be found in [6].

Beyond Global vs. Local Granularity

The dynamics of a robot swarm can be described from several points of view. The semantics of the `swarm` construct overcome the global vs. local granularity problem by allowing the developer to dynamically define the set of robots that should execute a certain portion of code. In the following, we describe four major approaches to the definition of swarm behaviors, showing how `swarm` can be used to express them.

Structure: Swarm-level Finite State Machine

In a seminal paper, Martinoli *et al.* [8] showed how swarm dynamics can be described through a finite state machine (FSM) in which each state is marked with the fraction of robots currently in that state. The swarm dynamics can then be described as a flow of robots that transition from state to state. A state typically corresponds to an individual robot behavior under execution. By describing the dynamics of a swarm through an FSM, the focus of the developer parts from the individual robot, to lay on the sequence of behaviors to execute and on the transitions among them.

The `swarm` constructs captures this concept naturally. Each state of the FSM is mapped to a dedicated `swarm`, while the transition logic is realized through calls to `swarm.unselect()` or `swarm.leave()`, and `swarm.select()` or `swarm.join()`. Figure 1 shows an example in which robots start in *Behavior 1*. Upon completing the execution of the behavior, a robot switches to *Idle*, waiting for other robots to finish. As soon as the all the robots are *Idle*, they switch to *Behavior 2*. The code also shows how the `stigmergy` construct can be used to implement a simple barrier to wait for all robots to be done with a behavior: each robot stores its id in the hash table, and when the size of the hash table matches the size of the swarm, the barrier is lifted.

Coordination: Distributed Task Allocation

Two recent works pioneered a new approach to distributed robot programming. Rather than concentrating on the robots, in Karma [9] and Voltron [10] the focus is on the tasks that the swarm must perform. The developer, in other words, specifies only the logic associated to a task, ignoring which robot will eventually execute it. The task are geographically distributed and assumed executable at any time. A dedicated run-time framework assigns robots to tasks dynamically, in a transparent fashion from the point of view of the developer. Both Karma and Voltron target single-robot single-task sensory scenarios, in which the robots move and observe the environment; however, the two languages differ in the way the tasks are coordinated.

The `swarm` construct offers a way to obtain a similar result, and to maintain full control on the process. The core of the idea is to structure the Buzz script in two macro-components: the dispatcher and the tasks. The role of the dispatcher is to keep track of the completion of the tasks, and to manage the transition of the robots from a task to another; each task corresponds to a dedicated swarm, and the logic associated to a task is a suitably defined Buzz function. A simple example of this approach is reported in Figure 2.

The advantage of structuring the code in this way is that task dispatching and task logic are decoupled, making it possible to modify, improve, and reuse a component without affecting the other. This is particularly useful to foster meaningful comparisons among distributed task dispatching algorithms, and to allow more ambitious research towards decentralized planning of interdependent tasks.

```
# Number of robots in the swarm, assumed known and fixed for simplicity
SWARM_SIZE = 100
# A numeric id for the barrier stigmergy
BARRIER_VSTIG_ID = 1
# Function to wait for everybody to be ready
function barrier_wait() {
  if(barrier.size() < SWARM_SIZE) barrier.get(id);
}
# Swarm states
STATE_BHVR1 = 1
STATE_IDLE  = 2
STATE_BHVR2 = 3
# This function is executed at initialization.
function init() {
  # Create swarms for each state
  bhvr1 = swarm.create(STATE_BHVR1)
  idle  = swarm.create(STATE_IDLE)
  bhvr2 = swarm.create(STATE_BHVR2)
  # Initially every robot is doing behavior 1
  bhvr1.join()
}
# Logic of behavior 1
function do_bhvr1() {
  var done = 0
  ...
  # We assume 'done' contains 1 if the behavior is finished
  if(done) {
    # Switch to idle state
    bhvr1.leave()
    idle.join()
    # Create a barrier using a stigmergy structure
    barrier = stigmergy.create(BARRIER_VSTIG)
    barrier.put(id, 1)
  }
}
# Logic of idle state
function do_idle() {
  # Is everybody done?
  if(barrier.size() == SWARM_SIZE) {
    # Switch to behavior 2
    idle.leave()
    bhvr2.join()
  }
}
# Logic of behavior 2
function do_bhvr2() {
  ...
}
# This function is executed at each time step.
function step() {
  bhvr1.exec(dobhvr1)
  idle.exec(doidle)
  bhvr2.exec(dobhvr2)
}
```

Figure 1: An example of a finite state machine implemented in Buzz. This code also shows how to use the `stigmergy` construct to implement a simple barrier.

Heterogeneity: Feature Tagging

An important aspect of future robot swarms is that their composition will be highly heterogeneous. From the hardware point of view, the benefits of heterogeneity stem from *(i)* the low cost of producing large quantities of specialized robots, rather than a small number of generic ones; and *(ii)* the possibility to deploy task-specific robots over time, lowering the cost and the complexity of swarm deployment. Other sources of heterogeneity are information (e.g., some robots are aware of a certain fact about the environment, others are not) and location (e.g., being in range to engage a target).

The `swarm` construct can be used to tag robots belonging to a certain category. Hardware heterogeneity is detectable through the presence or absence of specific language symbols. For instance, the BVM installed on a quad-rotor is likely to contain the definition of the function `fly_to()`, while a ground robot would have `set_wheel_speed()`. The specific choice on which symbols are present on which robot is done when designing the integration of the BVM on the robots, and it is (at least partially) application-specific. Figure 3 shows an example of swarm creation based on hardware heterogeneity in a swarm composed of quad-rotors and two types of ground-based robots, one of which is equipped with a manipulator. Information heterogeneity can be captured by creating swarms with conditions related to specific variables. Analogously, location heterogeneity can be dealt with by through conditions dependent from sensor readings.

Coherence: Team Tagging

Large robot swarms are often divided in teams which must operate as a single entity. These teams typically achieve spatial coherence by aggregation, flocking, or self-assembly, before moving towards the designated area for their mission. An important problem in this scenario is how to maintain team coherence when two teams operate in close proximity. For instance, when two flocks meet, the robots must be capable of recognizing similar (kin) robots and maintain formation with them, while avoiding strangers.

The `swarm` construct offers a solution to this problem. As previously explained, when a swarm lambda is executed a robot is aware of the swarms it belongs to. Some Buzz primitives, such as `neighbors` and `stigmergy`, internally use the swarm stack to limit their scope or to perform particular operations. In Figure 4 we report an example in which two swarms interact. The robots that belong to the same swarm maintain a short distance between each other, while keeping a longer distance to strangers. The core of the algorithm is in the `neighbors.kin()` and `neighbors.nonkin()` methods. The first method returns a list of robots belonging to the same swarm as the current robot, while the second returns a list of stranger robots.

Conclusions

From drones to self-driving cars, autonomous robotics systems are becoming pervasive, and are acting as an enabling technology for many kinds of safety-critical applications. Examples of robotic applications are search-and-rescue operations, industrial and agricultural inspection, autonomous car driving, aerial mapping, monument digitization, and surgery. Many of these applications require multiple robots to interact effectively because of the large-scale and dynamic nature of the environment in which the robots act. We envision a world in which a developer can specify the behavior of heterogeneous swarms of robots, and package this behavior in an application that can be installed on multiple robotic systems. There will be a market for these robotic *apps*, and manufacturers of robots and other devices will allow customers to purchase and customize these behaviors for their specific activities.

Among the challenges to overcome towards the realization of this vision, programmability is one of the most critical. In this article, we presented a language construct, called *swarm*, that allows developers to structure complex swarm algorithms into well-defined, reusable components. The main merit of the `swarm` construct is to provide a simple mechanism to overcome the global vs. local granularity choice.

Future work on this topic includes the creation of an algebra that captures the main features of this concept, and that allows to make sound predictions on the behavior of a swarm before execution.

References

[1] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," *Swarm Intelligence*, vol. 7, no. 1, pp. 1–41, Jan. 2013.

[2] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, 2009, p. 5.

[3] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous-space programs for robotic swarms," *Neural Computing and Applications*, vol. 19, no. 6, pp. 825–847, May 2010.

[4] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *ACM Symposium on Applied Computing*. ACM New York, 2015.

[5] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell, "A language for large ensembles of independently executing nodes," in *Logic Programming*, ser. LNCS 5649, P. M. Hill and D. S. Warren, Eds., vol. 5649 LNCS. Springer Berlin Heidelberg, 2009, pp. 265–280.

[6] C. Pinciroli, A. Lee-Brown, and G. Beltrame, "Buzz: An extensible programming language for self-organizing heterogeneous robot swarms," Available online at http://arxiv.org/abs/1507.05946, 2015.

[7] ——, "A tuple space for data sharing in robot swarms," in *9th EAI International Conference on Bio-inspired Information and Communications Technologies (BICT 2015)*. ACM Digital Library, 2015.

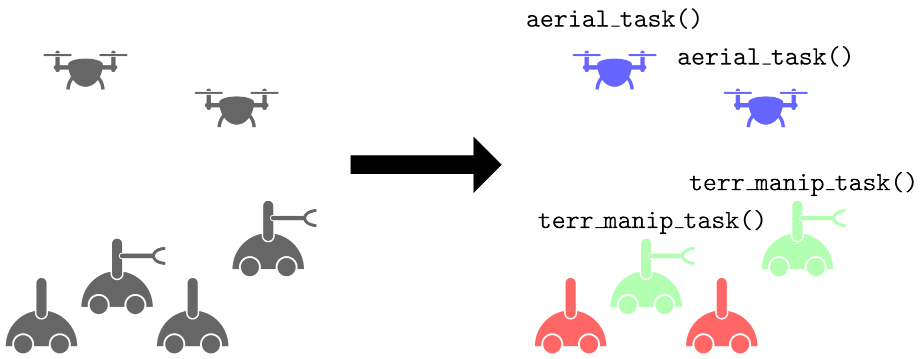
[8] A. Martinoli, K. Easton, and W. Agassounon, "Modeling Swarm Robotic Systems: a Case Study in Collaborative Distributed Manipulation," *The International Journal of Robotics Research*, vol. 23, no. 4, pp. 415–436, apr 2004.

[9] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh, "Programming micro-aerial vehicle swarms with Karma," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems - SenSys '11*. New York, New York, USA: ACM Press, 2011, pp. 121–134.

[10] L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi, "Team-level Programming of Drone Sensor Networks," in *SenSys '14 Proceedings of the 12th ACM Conference on Embedded Network Sensor SystemsSystems*. ACM New York, NY, 2014, pp. 177–190.

```
# Total number of tasks to perform
TASKNUM = 2
# This function is executed at initialization
function init() {
  # Create a swarm for each task
  idle  = swarm.create(1)
  task1 = swarm.create(2)
  task2 = swarm.create(3)
  # Initially every robot is idle
  idle.join()
  # Create a stigmergy to store task completion status
  taskcomp = stigmergy.create(1)
  taskcomp.put(1, task1)
  taskcomp.put(2, task2)
}
# A simple task dispatcher that assigns to a robot the first
# available task
function task_dispatch() {
  # Check if more tasks are available
  if(taskcomp.size() > 0) {
    # Pick the first uncompleted task
    for(var i = 0; i < TASKNUM; i = i + 1) {
      if(taskcomp.get(i)) {
        # Switch to task
        taskcomp.get(i).join()
        # Remove task from pool
        taskcomp.put(i, nil)
      }
    }
  }
}
# Idle robot logic
function do_idle() {
  task_dispatch()
}
# Task 1 logic
function do_task1() {
  var done = 0
  # Logic of task 1
  ...
  if(done) {
    # Switch to idle
    task1.leave()
    idle.join()
  }
}
# Task 2 logic
function do_task2() {
  var done = 0
  # Logic of task 2
  ...
  if(done) {
    # Switch to idle
    task2.leave()
    idle.join()
  }
}
# This function is executed at each time step
function step() {
  idle.exec(do_idle)
  task1.exec(do_task1)
  task2.exec(do_task2)
}
```

Figure 2: A simple task dispatcher in Buzz. The code is divided in two components: the task dispatcher, encoded in function `task_dispatch()`, and the actual tasks. Each task is expressed through a `swarm`, which executes the corresponding logic encoded in the `do_task*()` functions. The stigmergy structure is used to keep track of the completed tasks.



```
# Group identifiers
AERIAL          = 1
TERRESTRIAL     = 2
MANIPULATORS    = 4
# Task for aerial robots
function aerial_task() { ... }
# Task for terrestrial manipulator robots
function terr_manip_task() { ... }
# Create swarm with robots possessing the 'fly_to' symbol
aerial = swarm.create(AERIAL)
aerial.select(fly_to)
# Create swarm with robots possessing the 'set_wheel_speed' symbol
terrestrial = swarm.create(TERRESTRIAL)
terrestrial.select(set_wheel_speed)
# Create swarm with robots possessing the 'grip' symbol
manipulators = swarm.create(MANIPULATORS)
manipulators.select(grip)
# Assign task to terrestrial manipulators
terr_manip = swarm.intersection(TERRESTRIAL + MANIPULATORS,
                                terrestrial, manipulators)
terr_manip.exec(aerial_task)
# Assign task to aerial robots
aerial.exec(terr_manip_task)
```

Figure 3: Through `swarm` variables, the developer can create swarms and assign them tasks. In this example, three `swarm`s are created, and two of them are assigned tasks to execute.

```
# Constants
TARGET_KIN     = 100.
EPSILON_KIN    = 100.
TARGET_NONKIN  = 300.
EPSILON_NONKIN = 250.
# Lennard-Jones interaction magnitude
function calc_lj(dist, target, epsilon) {
  return -(epsilon / dist) * ((target / dist)^4 - (target / dist)^2)
}
# Neighbor data to kin LJ interaction
function to_lj_kin(rid, data) {
  var lj
  lj = calc_lj(data.distance, TARGET_KIN, EPSILON_KIN)
  data.x = lj * math.cos(data.azimuth)
  data.y = lj * math.sin(data.azimuth)
  return data
}
# Neighbor data to non-kin LJ interaction
function to_lj_nonkin(rid, data) {
  var lj
  lj = calc_lj(data.distance, TARGET_NONKIN, EPSILON_NONKIN)
  data.x = lj * math.cos(data.azimuth)
  data.y = lj * math.sin(data.azimuth)
  return data
}
# Accumulator of neighbor LJ interactions
function vec2_sum(rid, data, accum) {
  accum.x = accum.x + data.x
  accum.y = accum.y + data.y
  return accum
}
# Actual flocking logic
function flock() {
  # Create accumulator
  var accum = {}
  accum.x = 0
  accum.y = 0
  # Calculate accumulator
  accum = neighbors.kin().map(to_lj_kin).reduce(vec2_sum, accum)
  accum = neighbors.nonkin().map(to_lj_nonkin).reduce(vec2_sum, accum)
  if(neighbors.count() > 0) {
    accum.x = accum.x / neighbors.count()
    accum.y = accum.y / neighbors.count()
  }
  # Move according to vector
  goto(accum.x, accum.y)
}
```

Figure 4: An example Buzz script to define the spatial interaction among swarms. Through the `neighbors.kin()` and `neighbors.nonkin()` methods, it is possible to encode a logic that maintains a short distance between robots in the same swarm, and a long distance between robots in different swarms.