2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
Daejeon Convention Center
October 9-14, 2016, Daejeon, Korea

Buzz: An Extensible Programming Language for Heterogeneous Swarm Robotics

Carlo Pinciroli *Member, IEEE* and Giovanni Beltrame *Member, IEEE*

Abstract— **We present Buzz, a novel programming language for heterogeneous robot swarms. Buzz advocates a compositional approach, offering primitives to define swarm behaviors both from the perspective of the single robot and of the overall swarm. Single-robot primitives include robot-specific instructions and manipulation of neighborhood data. Swarm-based primitives allow for the dynamic management of robot teams, and for sharing information globally across the swarm. Self-organization stems from the completely decentralized mechanisms upon which the Buzz run-time platform is based. The language can be extended to add new primitives (thus supporting heterogeneous robot swarms), and its run-time platform is designed to be laid on top of other frameworks, such as the Robot Operating System. We showcase the capabilities of Buzz by providing code examples, and analyze scalability and robustness of the run-time platform through realistic simulated experiments with representative swarm algorithms.**

I. Introduction

Swarm robotics systems [1] are envisioned for large-scale application scenarios that require reliable, scalable, and autonomous behaviors. Among the many hurdles towards real-world deployment of swarm robotics systems, one of the most important ones is the lack of dedicated tools, especially regarding software [2]. In particular, one problem that has received little attention in the literature is *programmability*. The current practice of swarm behavior development focuses on individual behaviors and low-level interactions [3]. This approach forces developers to constantly 'reinvent the wheel' to fit well-known algorithms into new applications, resulting in a slow and error-prone development process.

To promote code reuse, two general approaches are possible. The first approach is the development of software libraries that encapsulate certain algorithms. While this has the advantage of leveraging pre-existing frameworks and tools, it falls short of screening the user from unnecessary detail, such as non-trivial compilation configuration or language-specific boilerplate. The second approach is the creation of a domain-specific language (DSL) exposing only the relevant aspects of the problem to solve. A well-designed DSL shapes the way a system is conceived, by offering powerful abstractions expressed through concise constructs.

In this paper, we argue that a DSL is an indispensable tool towards the deployment of real-world robot swarms. The dynamics of robot swarms are made complex by the presence of both spatial aspects (body shape, sensing, actuation) and network aspects (message loss, volatile topology). Additionally, to render the production of large-scale swarms affordable, the robots suffer from significant limitations in terms of computational power, especially when compared with robots designed to act alone. Thus, a DSL for robot swarms has the potential to act as a platform that *(i)* Filters the low-level details concerning space and networking, in an efficient, resource-aware fashion; *(ii)* Offers a coherent abstraction of the system; *(iii)* Acts as common platform for benchmarking, code reuse, and comparison—a prerequisite for the creation of solid 'swarm engineering' practices.

In this paper, we present Buzz, a novel DSL for robot swarms. To drive the design of Buzz, we identified key requirements a successful programming language for swarm robotics must meet. First, as mentioned, the language must allow the programmer to work at a suitable level of abstraction. The complexity of concentrating on individual robots and their interactions, i.e., a *bottom-up* approach, increases steeply with the size of the swarm. Conversely, a purely *top-down* approach, i.e., focused on the behaviour of the swarm as a whole, might lack expressive power to fine-tune specific robot behaviors. We believe that a language for robot swarms must combine *both* bottom-up and top-down primitives, allowing the developer to pick the most comfortable level of abstraction to express a swarm algorithm. Second, the language must enable a compositional approach, by providing predictable primitives that can be combined intuitively into more complex constructs. Third, the language must prove generic enough to *(i)* express the most common algorithms for swarm coordination, such as flocking, task allocation, and collective decision making; and *(ii)* support heterogeneous swarms. Fourth, the run-time platform of the language must ensure acceptable levels of scalability (for increasing swarm sizes) and robustness (in case of temporary communication issues). Currently, the management of the low-level aspects concerning these issues constitutes a sizable portion of the development process of swarm behaviors. Alleviating this burden with a scalable and robust run-time platform is crucial for real-world deployment of swarm behaviors.

The main contribution of this paper is the design and implementation of Buzz, a programming language that meets the requirements discussed above. Buzz is released as open-source software under the MIT license. It can be downloaded at `http://the.swarming.buzz/`.

II. Design Principles

The design of Buzz was based on a set of high-level principles, that are described in the following.

Carlo Pinciroli and Giovanni Beltrame are with the Department of Computer and Software Engineering, École Polytechnique de Montréal, Montréal, Canada. E-mail: `{carlo.pinciroli,giovanni.beltrame}@polymtl.ca`

a) Discrete swarm, step-wise execution: A swarm is considered as a discrete collection of devices, each running the Buzz virtual machine (BVM), and executing the same Buzz script. Script execution proceeds independently on each robot in a step-wise fashion. Each time-step is divided in five phases: 1) Sensor readings are collected and stored in the BVM; 2) Incoming messages are collected and processed by the BVM; 3) A portion of the Buzz script is executed; 4) The messages in the BVM output queue are sent (as many as possible, according to the available payload size); 5) Actuator values are collected from the BVM state and applied. The length of a time-step is constant over the life-time of the script, although it does not need to be the same for every robot. If the execution time of a step is shorter than the allotted time, the robot sleeps for a period. If the execution is longer, a warning message is issued (e.g., printed on the screen of the developer monitoring the swarm).

b) Predictable and composable syntax: Buzz employs a syntax that mixes imperative and functional constructs, inspired by languages such as JavaScript, Lua, and Python. While this syntax ensures a short learning curve, it also allows for predictable and composable scripts. By *predictable*, we mean that a programmer can easily infer the results of the execution of a program by reading its code. *Composability* refers to the ability to structure a complex program into simpler elements, such as functions, classes, etc. that can later be used as modules.

c) Support for heterogeneous robots: Buzz is explicitly designed to support heterogeneous robot swarms. To this aim, Buzz is conceived as an *extension language*, that is, a language that exports and enhances the capabilities of an already existing system. This design choice allows one to stack the Buzz run-time on top of well-known single-robot frameworks such as ROS [4],[1] OROCOS,[2] or YARP.[3] The Buzz syntax and primitives are minimal, concise, and powerful; the programmer can add new primitives and constructs that link to relevant features of the underlying system.

d) Swarm-level abstraction: One of the novel aspects of Buzz is the ability to manage swarms of robots. The concept of *swarm* is a first-class language object. Swarms can be created and disbanded; new swarms can be created as a result of an operation of intersection, union, or difference between two pre-existing swarms. Each swarm can be atomically assigned operations to perform. Swarms can be employed to encode complex task coordination strategies.

e) Situated communication: Each robot is assumed to be equipped with a device capable of *situated communication* [5]. Such device broadcasts messages within a limited range and receives messages from neighboring robots in direct, unobstructed line-of-sight. Upon receipt of a message, a robot also detects the relative distance and angle of the message sender. Situated communication has been used extensively in swarm robotics to achieve global coordination in algorithms for, e.g., pattern formation [6], flocking [7], exploration [8], and task allocation [9]. This communication modality can be achieved on-hardware with robots such as the Kilobot [10], the e-puck [11], and the marXbot [12], although with limited payload size, and a dedicated module for low-cost 3D communication was proposed in [13]. Alternatively, situated communication can be satisfactorily realized through WiFi and GPS (for outdoor scenarios) or tracking systems such as Vicon (for indoor scenarios).

f) Information sharing: Buzz provides two ways for robots to share information: virtual stigmergy and neighbor queries. *Virtual stigmergy* [14] is a data structure that allows the robots in a swarm to globally agree on the values of a set of variables. From an abstract point of view, virtual stigmergy is akin to a distributed key-value storage. Among its many potential applications, this structure may be employed to coordinate task execution by, e.g., marking the tasks that have already been completed and/or sharing progress on the uncompleted ones. *Neighbor queries* allow a robot to inquire its direct neighbors on the value of a certain variable. Upon receiving a request, the neighbors reply with the current value of the variable; these replies are then collected into a list by the robot who requested the data. Neighbor queries are useful when robots must perform local data aggregation, such as calculating averages [15] or constructing spatial gradients.

III. Language Definition

The Backus-Naur Form language grammar is available at `http://the.swarming.buzz/wiki/doku.php?id=buzz_syntax_bnf_specification`.

A. Robot-wise operations and primitive types

The simplest operations available in Buzz are robot-wise operations. These operations are executed by every robot individually. Assignments, arithmetic operations, loops and branches fall into this category.

Buzz is a dynamically typed language that offers the following types: `nil`, integer, floating-point, string, table, closure, swarm, virtual stigmergy, and user data. The `nil`, integer, floating-point, and string types work analogously to other scripting languages such as JavaScript, Python, or Lua. The swarm and virtual stigmergy types are introduced later on in the paper in dedicated sections.

Tables are the only structured type available in Buzz, and they are inspired by the analogous construct in Lua [16]. Tables can be used as arrays or dictionaries.

Buzz also supports functions as first-class objects (*lambdas*), implemented as closures. Two types of closures exist: native closures and C closures. Native closures are functions defined within a Buzz script, while C closures are C functions registered into the BVM (which is written in C).

Because Buzz is an extension language, it is likely for the compiler to encounter unknown symbols. In case the symbol is unknown at run-time, its value is set to `nil`. This typically occurs when robots with different capabilities are employed. For instance, flying robots may provide a `fly_to()` command for motion, while wheeled robots may

[1]http://www.ros.org/
[2]http://www.orocos.org/
[3]http://wiki.icub.org/yarp/

provide a `set_wheels()` command. As shown in the next section, this mechanism provides a flexible way to write scripts that can work on robots with diverse capabilities.

B. Swarm management

Buzz lets the programmer subdivide the robots into multiple teams. In Buzz parlance, each team is called a *swarm*. Buzz treats swarms as first-class objects. To create a swarm, the programmer must provide a unique identifier, which is known and shared by all the robots in the created swarm. The returned value is a structure of type `swarm`.

Once a swarm is created, it is initially empty. To have robots join a swarm, two methods are available: `select()` and `join()`. With the former, the programmer can specify a condition, evaluated by each robot individually, for joining a swarm; with the latter, a robot joins a swarm unconditionally. To leave a swarm, Buzz offers the analogous methods `unselect()` and `leave()`. A robot can check whether it belongs to a swarm through the method `in()`.

The programmer can create new swarms that result from operations on pre-existing swarms. Four such operations are available: intersection, union, difference, and negation.

Through the method `exec()`, the developer can assign tasks to specific swarms. A simple example of task allocation in a heterogenous swarm is reported:

```
# Group identifiers
AERIAL   = 1
GROUND   = 2
GRIPPERS = 4
# Task for aerial robots
function aerial_task() { ... }
# Task for ground-based gripper robots
function ground_gripper_task() { ... }
# Create swarm of robots with 'fly_to' symbol
aerial = swarm.create(AERIAL)
aerial.select(fly_to)
# Create swarm of robots with 'set_wheels' symbol
ground = swarm.create(GROUND)
ground.select(set_wheels)
# Create swarm of robots with 'grip' symbol
grippers = swarm.create(GRIPPERS)
grippers.select(grip)
# Assign task to aerial robots
aerial.exec(aerial_task)
# Assign task to ground-based gripper robots
ground_grippers = swarm.intersection(GROUND + GRIPPERS, ground, grippers)
ground_grippers.exec(ground_gripper_task)
```

C. Neighbor operations

Buzz offers a rich set of operations based on the neighborhood of a robot. These operations include both spatial and communication aspects. The entry point of all neighbor operations is the `neighbors` structure. For each robot, this structure stores spatial information on the neighbors within communication range. The structure is updated at each time step. It is internally organized as a dictionary, in which the index is the id of the neighbor and the data entry is a tuple `(distance, azimuth, elevation)`.

a) Iteration, transformation, reduction: The `neighbors` structure admits three operations: iteration, transformation, and reduction. Iteration, encoded in method `foreach()`, allows the programmer to apply a function without return value to each neighbor. Transformation, encoded in method `map()`, applies a function with return value to each neighbor. The return value is the result of an operation on the data associated to a neighbor. The result of mapping is a new `neighbors` structure, in which each neighbor id is associated to the transformed data entries. Reduction, encoded in method `reduce()`, applies a function to each neighbor to produce a single result.

b) Filtering: It is often useful to apply the presented operations to a subset of neighbors. The `filter()` method allows the programmer to apply a predicate to each neighbor. The end result of the `filter()` method is a new `neighbors` structure storing the neighbors for which the predicate (a function) was true. Another common necessity is filtering neighbors by their membership to a swarm. The `kin()` method returns a `neighbors` structure that contains the robots that belong to the same top-of-the-stack swarm as the current robot. The `nonkin()` method returns the complementary structure. An example application for these methods is offered in the example section.

The `neighbors` structure provides a natural way to implement pattern formation behaviors, as shown in this example of formation of a square lattice.

```
# Virtual force magnitude
function force_mag(dist, delta, epsilon) {
  return -(epsilon / dist) * ((delta / dist)^4 - (delta / dist)^2)
}
# Virtual force sum functions
function forcesum_kin(rid, data, accum) {
  var fm = force_mag(data.distance, DELTA_K, EPS_K)
  accum.x = accum.x + fm * math.cos(data.azimuth)
  accum.y = accum.y + fm * math.sin(data.azimuth)
  return accum
}
function forcesum_nonkin(rid, data, accum) {
  var fm = force_mag(data.distance, DELTA_NK, EPS_NK)
  accum.x = accum.x + fm * math.cos(data.azimuth)
  accum.y = accum.y + fm * math.sin(data.azimuth)
  return accum
}
# Calculates the direction vector
function direction() {
  var dir = neighbors.kin().reduce(forcesum_kin, {.x=0,.y=0})
  dir = neighbors.nonkin().reduce(forcesum_nonkin, dir)
  dir.x = dir.x / neighbors.count()
  dir.y = dir.y / neighbors.count()
  return dir
}
# Executed at each step
function step() { goto(direction()) }
```

c) Communication: Another use for the `neighbors` structure is to exchange and analyze local data. To make this possible, Buzz offers two methods: `broadcast()` and `listen()`. The former allows the robot to broadcast a `(key, value)` pair across its neighborhood. The latter takes two inputs: the `key` to listen to, and a `listener` function to execute upon receiving a value from a neighbor. The BVM executes the `listener` whenever data is received until the method `neighbors.ignore(key)` is called. As an example, we report a script in which a swarm forms a distance gradient from a robot acting as source.

```
# Define function to execute upon receiving information on
# the distance to the target
function init() {
  neighbors.listen("dist_to_target",
    function(topic, value, robot_id) {
      dist = math.min(dist, neighbors.get(robot_id).distance + value)})
}
# This function is executed at each time step.
function step() {
  # Broadcast currently known distance to target
  neighbors.broadcast("dist_to_target", dist)
}
```

D. Virtual stigmergy

Virtual stigmergy [14] is a data structure that allows a swarm of robots to share data globally. Essentially, virtual stigmergy works as a distributed tuple space analogous to Linda [17]. Three methods are available: `create()`, put(), and get(). As the names suggest, create() is a method that creates a new virtual stigmergy structure, while put() and get() access the structure, writing or reading (key, value) entries. Virtual stigmergy handles only a subset of the primitives types: integer, floating-point, string, and table. These types can be used either as keys or values.

The name *virtual stigmergy* derives from the indirect, environment-mediated communication of nest-building insects such as ants and termites [18]. The key idea in natural stigmergy is that environmental modifications to organize the environment occur in a step-wise fashion, whereby a modification performed by an individual causes a behavioral response in another individual, without the two individuals ever directly interacting. From the point of view of the programmer, virtual stigmergy works as a virtual shared environment: a robot modifies an entry, and this modification triggers a reaction in another robot without the two having to interact directly.

Virtual stigmergy is a powerful concept that enables the implementation of a large class of swarm behaviors. As an example, we show how to express a *barrier*—a synchronization mechanism that allows a swarm to wait until a sufficient number of robots are ready to proceed. Quorum sensing has been proposed as a way to achieve this behavior [19]. Virtual stigmergy allows one to implement this mechanism. Every time a robot wants to mark itself as ready, it puts a pair (id, 1) in the virtual stigmergy structure, where id is a robot's numeric id. The size() method returns the number of tuples in the virtual stigmergy, which corresponds to the count of ready robots. When this count reaches the swarm size (or another application-dependent threshold value), the robots collectively trigger the next phase.

```
# A numeric id for the barrier virtual stigmergy
BARRIER_VSTIG = 1
# Function to mark a robot ready
function barrier_set() {
  barrier = stigmergy.create(BARRIER_VSTIG)
}
# Function to mark a robot ready
function barrier_ready() {
  barrier.put(id, 1)
}
# Function to wait for everybody to be ready
function barrier_wait(threshold) {
  while(barrier.size() < threshold) barrier.get(id);
}
```

IV. Run-Time Platform

A. The Buzz virtual machine

The run-time platform of Buzz is based on a custom, stack-based virtual machine written entirely in C. The organization of the modules composing the VM is depicted in Figure 1. The implementation of a new VM is motivated by the design choices in Buzz. In particular, the integration of swarm management and virtual stigmergy into a VM for a dynamically-typed, extensible language forced us to find dedicated solutions for data representation, stack/heap management, and byte code encoding. These aspects are purely technical and go beyond the scope of this paper. A notable fact about the BVM is its tiny size (12 KB) which fits most robots currently in use for swarm robotics research.

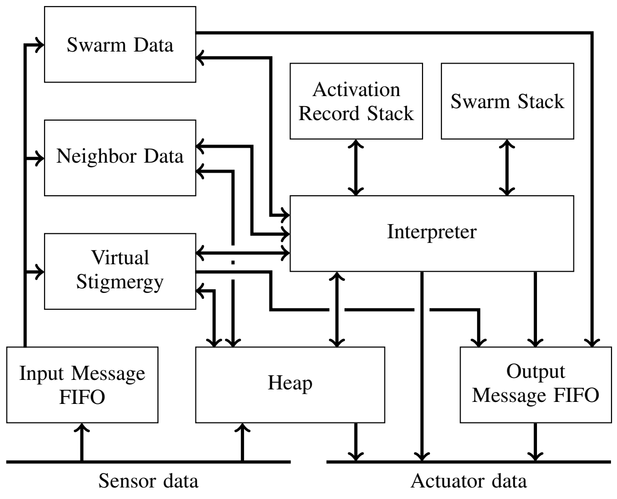


Fig. 1. The structure of the Buzz virtual machine.

At each time step, the BVM state is updated with the latest sensor readings. Sensor readings are typically stored in the heap as data structures (e.g., tables). Incoming messages are then inserted in the BVM, which proceeds to unmarshal them and update the relevant modules. Subsequently, the interpreter is called to execute a portion of the script. At the end of this phase, the updated actuator data is read from the BVM heap and part of the messages in the outbound queue are sent by the underlying system.

Further details on the low-level implementation of the BVM are reported in [20].

B. Code Compilation

The compilation of a Buzz script involves two tools: a compiler called buzzc and an assembler/linker called buzzasm. The former is a classical recursive descent parser that generates an annotated object file in a single pass. The latter parses the object file, performs the linking phase, and generates the byte code. The compilation process is typically performed on the programmer's machine, and the generated byte code is then uploaded on the robots.

V. Experimental Evaluation

In this section, we analyze the scalability and robustness of the Buzz run-time. We focus our analysis on two subsystems, virtual stigmergy and neighbor queries, because of their key role in swarm coordination. The evaluation is conducted through simulations in ARGoS [21] with the ground-based marXbot robot [12].

a) Experimental setup: Our experimental setup consists of a square arena of side L in which N robots are randomly scattered. The coordinates (x, y) of each robot are chosen at random with a uniform distribution from $\mathcal{U}(-L/2, L/2)$.[4] We define the robot density D as the ratio between the area occupied by all the robots and the total area of the arena. To ensure comparable conditions across different choices of

[4]When a coordinate choice causes physical overlap with already placed robots, a new coordinate is picked until no overlap occurs.

N, we keep the density constant ($D = 0.1$) and calculate L with:

$$D = \frac{N\pi R^2}{L^2} \Rightarrow L = \sqrt{\frac{N\pi R^2}{D}},$$

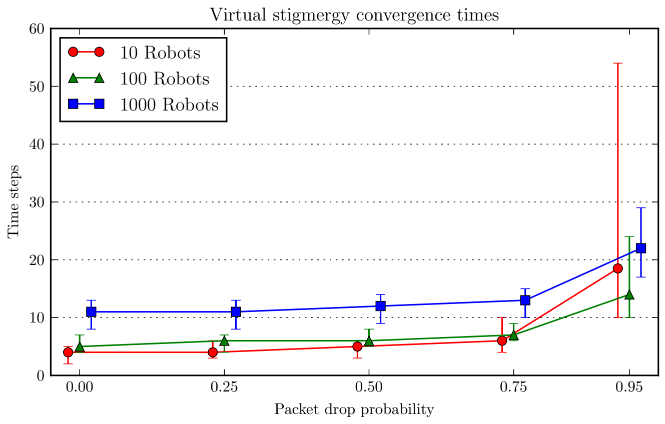
where $R = 8.5\,\mathrm{cm}$ is the radius of a marXbot. We focus our analysis on two parameters that directly affect the properties that we intend to analyze: *(i)* The number of robots N, which impacts scalability; and *(ii)* The message dropping probability P, which affects robustness and accounts for an important, unavoidable phenomenon that influences the efficiency of current devices for situated communication. For N, we chose $\{10, 100, 1000\}$; for P, we chose $\{0, 0.25, 0.5, 0.75, 0.95\}$. Each experimental configuration $\langle N, P\rangle$ was tested 100 times.

b) Virtual stigmergy: To analyze the efficiency of virtual stigmergy, we devised an experiment in which the robots must agree on the highest robot id across the swarm. This experiment is representative of a wide class of situations in which a robot swarm must agree on the maximum or minimum value of a quantity (e.g., sensor reading). As performance measure, we employed the number of time steps necessary to reach global consensus. We report in Figure 2 the script we executed and the data distribution we obtained. Our results indicate that, up to $P = 0.75$, the number of time steps necessary to reach consensus is affected weakly by N, and practically unaffected by P. Interestingly, for $\langle N = 1000, P = 0.75\rangle$, consensus is reached in at most 15 time steps (a time step corresponds to $0.1\,\mathrm{s}$ in our simulations). This positive result can be explained by noting that, with $P = 0.75$, 3 messages out of 4 are lost; however, the uniform distribution of the robots ensures that, on average, each robot has more than 3 neighbors. Thus, messages can still flow throughout the network. The effect of packet dropping is apparent for very pessimistic values ($P = 0.95$). A more thorough study on the effect of topology and motion on the performance of virtual stigmergy is in [14].

c) Neighbor queries: To analyze the performance of neighbor queries, we devised an experiment in which the robots must construct a distance gradient from a robot acting as source. This experiment is representative because the formation of gradients is a fundamental coordination mechanism in swarm behaviors. As performance measure, we employ the time necessary for every robot to estimate its distance to the source. The script and the data plot are reported in Figure 3. The dynamics are analogous to virtual stigmergy: convergence time depends weakly on N and is unaffected by P up to $P = 0.75$; for $\langle N = 1000, P = 0.75\rangle$ convergence is reached in a maximum of 13 time steps; for $P = 0.95$, convergence times increase sensibly. Again, this is arguably due to the dense distribution of robots, which facilitates the circulation of messages across the swarm, thus mitigating the effects of message dropping.

VI. Related Work

Most of the literature in programming languages designed for robotics focus on individual robots. Prominent examples of this body of work are Willow Garage's Robot Operating



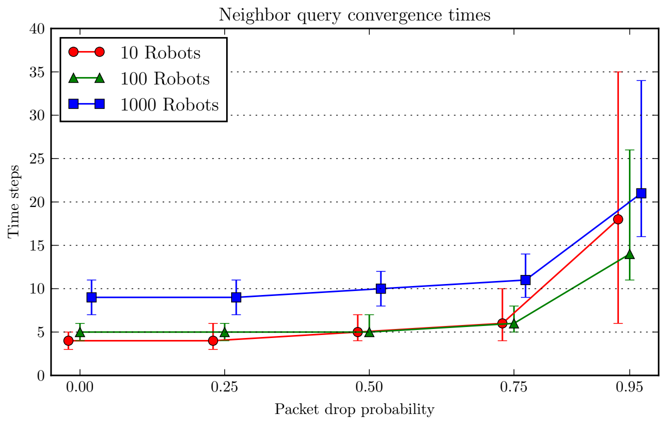
```
# Executed at init time
function init() {
  # Create a vstig
  VSKEY = 1
  vs = stigmergy.create(1)
  # Set onconflict manager
  vs.onconflict(function(k,l,r) {
    # Return local value if
    # - Remote value is smaller than local, OR
    # - Values are equal, robot of remote record is
    #   smaller than local one
    if(r.data < l.data or (r.data == l.data and r.robot < l.robot)) {
      return l
    }
    # Otherwise return remote value
    else return r
  })
  # Initialize vstig
  vs_value = id
  vs.put(VSKEY, vs_value)
}
# Executed at each time step
function step() {
  # Get current value
  vs_value = vs.get(VSKEY)
}
```

Fig. 2. Virtual stigmergy performance assessment. The plot reports the median, max, and min values of the distributions obtained for each experimental configuration $\langle N, P\rangle$. The markers are slightly offset to make them visible.

System (ROS) [4], and the event-based language URBI [22]. In swarm robotics, SWARMORPH-script [23] is a bottom-up behavior-based scripting language designed to achieve morphogenesis with mobile robots.

The last decade saw the first top-down approaches for the development of distributed computing systems. Various abstractions and programming languages have been proposed in the sensor network community [24]. A programming methodology inspired by embryogenesis and designed for self-assembly applications was proposed in [25]. Dantu *et al.* proposed Karma [26], a framework that combines centralized and distributed elements to perform task allocation in a swarm of aerial robots unable to communicate directly.

The spatial computing community has also produced a large body of work on programming languages [27]. In spatial computing, a distributed system is seen as a collection of connected computing devices scattered in a physical space. Proto [28] is arguably the most successful language of this kind. In Proto, the spatial computer is modeled as a continuous medium in which each point is assigned a tuple of values. The primitive operations of Proto act on this medium. The LISP-like syntax of Proto is modular by design and produces predictable programs. Proto shines in scenarios in which homogeneous devices perform distributed



```
# Constant for infinite distance (500 m)
DISTANCE_INF = 50000.
# Executed at init time
function init() {
  if(id == 0)
    # Source robot
    mydist = 0.
  else {
    # Other robots
    mydist = DISTANCE_INF
    # Listen to other robots' distances
    neighbors.listen("dist_to_source",
      function(vid, value, rid) {
        mydist = math.min(mydist, neighbors.get(rid).distance + value)
      })
  }
}
# Executed at each time step
function step() {
  neighbors.broadcast("dist_to_source", mydist)
}
```

Fig. 3. Neighbor query performance assessment. The plot reports the median, max, and min values of the distributions obtained for each experimental configuration $\langle N, P\rangle$. The markers are slightly offset to make them visible.

spatial computation—the inspiration for Buzz' `neighbors` construct was taken from Proto. However, as a language for robotics, Proto presents a number of limitations: *(i)* As a functional language, maintaining state over time is cumbersome; *(ii)* Every node handles only single tuples; *(iii)* Support for robot motion is limited; *(iv)* No explicit support for heterogeneous devices is present. Recently, Pianini *et al.* [29] proposed Protelis, a language that provides Proto-like abstractions as extensions of the Java language.

Meld [30] is a declarative language that realizes the top-down approach by allowing the developer to specify a high-level, logic description of what the swarm as a whole should achieve. The low-level (communication/coordination) mechanisms that reify the high-level goals, i.e., the how, are left to the language implementation and are transparent to the developer. The main concepts of the language are facts and rules. A fact encodes a piece of information that the system considers true at a given time. A computation in Meld consists of applying the specified rules progressively to produce all the true facts, until no further production is possible. Meld supports heterogeneous robot swarms by endowing each robot with facts that map to specific capabilities. A similar concept exists in Buzz, with robot-specific symbols. The main limitation of Meld for swarm robotics is the fact that its rule-based mechanics produce programs whose execution is difficult to predict and debug, and it is thus impossible to decompose complex programs into well-defined modules.

Voltron [31] is a language designed for distributed mobile sensing. Voltron allows the developer to specify the logic to be executed at several locations, without having to dictate how the robots must coordinate to achieve the objectives. Coordination is achieved automatically through the use of a shared tuple space, for which two implementations were tested—a centralized one, and a decentralized one based on the concept of virtual synchrony [32]. In Buzz, virtual stigmergy was loosely inspired by the capabilities of virtual synchrony, although the internals of the two systems differ substantially. Voltron excels in single-robot, single-task scenarios in which pure sensing is involved; however, fine-grained coordination of heterogeneous swarms is not possible, because Voltron's abstraction hides the low-level details of the robots.

VII. Conclusions and Future Work

We presented Buzz, a novel programming language designed for large-scale, heterogeneous robot swarms. The contributions of our work include: *(i)* a mixed paradigm for the implementation of robot swarms, which allows the developer to specify fine-grained, bottom-up logic as well as reason in a top-down, swarm-oriented fashion; *(ii)* the definition of a compositional and predictable approach to swarm behavior development; *(iii)* the implementation of a general language capable of expressing the most common swarm behaviors.

Future work on Buzz will involve several activities. Firstly, we will integrate the run-time into multiple robotics platforms of different kinds, such as ground-based and aerial robots. Secondly, we will create a library of well-known swarm behaviors, which will be offered open-source to practitioners as part of the Buzz distribution. This will be the first true collection of 'swarm patterns', in the classical sense the word 'pattern' assumes in software engineering [33]. Finally, we will tackle the design of general approaches to swarm behavior debugging and fault detection. These topics have received little attention in the literature, and Buzz constitutes an ideal platform to study them. In particular, we will study more in depth the impact of the network topology on the efficiency of message passing, and we will investigate adaptive methods to detect and mitigate the issues of unoptimal topologies.

Acknowledgments

This work was supported by an NSERC Engage Grant and an NSERC Discovery Grant. Computations were made on the supercomputer Mammouth-Ms from Université de Sherbrooke, managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), NanoQuébec, RMGA and the Fonds de recherche du Québec — Nature et technologies (FRQ-NT).

References

[1] G. Beni, "From Swarm Intelligence to Swarm Robotics," *Swarm Robotics*, vol. 3342, pp. 1–9, 2005.

[2] J. McLurkin, J. Smith, J. Frankel, D. Sotkowitz, D. Blau, and B. Schmidt, "Speaking Swarmish : Human-Robot Interface Design for Large Swarms of Autonomous Mobile Robots," in *AAAI Spring Symposium: To Boldly Go Where No Human-Robot Team Has Gone Before*. AAAI, 2006, pp. 3–6.

[3] 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.

[4] 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.

[5] K. Støy, "Using situated communication in distributed autonomous mobile robots," in *Proceedings of the 7th Scandinavian Conference on Artificial Intelligence*. IOS Press, 2001, pp. 44–52.

[6] W. M. Spears, D. F. Spears, J. C. Hamann, and R. Heil, "Distributed, Physics-Based Control of Swarms of Vehicles," *Autonomous Robots*, vol. 17, no. 2/3, pp. 137–162, Sept. 2004.

[7] E. Ferrante, A. E. Turgut, A. Stranieri, C. Pinciroli, M. Birattari, and M. Dorigo, "A self-adaptive communication strategy for flocking in stationary and non-stationary environments," *Natural Computing*, vol. 13, no. 2, pp. 225–245, 2014.

[8] C. Pinciroli, M. Bonani, F. Mondada, and M. Dorigo, "Adaptation and awareness in robot ensembles: Scenarios and algorithms," in *Software Engineering for Collective Autonomic Systems*, M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds. Springer International Publishing, 2015, vol. LNCS 8998, ch. IV.2, pp. 471–494.

[9] A. Brutschy, G. Pini, C. Pinciroli, M. Birattari, and M. Dorigo, "Self-organized task allocation to sequentially interdependent tasks in swarm robotics," *Autonomous Agents and Multi-Agent Systems*, vol. 28, no. 1, pp. 101–125, 2014.

[10] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," *2012 IEEE International Conference on Robotics and Automation*, pp. 3293–3298, May 2012.

[11] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, J.-C. Zufferey, D. Floreano, and A. Martinoli, "The e-puck , a Robot Designed for Education in Engineering," in *Proceedings of Robotica 2009 – 9th Conference on Autonomous Robot Systems and Competitions*, P. J. S. Gonçalves, P. J. D. Torres, and C. M. O. Alves, Eds., vol. 1. IPCB, Castelo Branco, Portugal, 2006, pp. 59–65.

[12] M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada, "The marXbot, a miniature mobile robot opening new perspectives for the collective-robotic research," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Piscataway, NJ: IEEE Press, 2010, pp. 4187–4193.

[13] O. De Silva, G. Mann, and R. Gosine, "An ultrasonic and vision-based relative positioning sensor for multirobot localization," *IEEE Sensors Journal*, vol. 15, no. 3, pp. 1716–1726, 2014.

[14] C. Pinciroli, A. Lee-Brown, and G. Beltrame, "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, in press.

[15] M. Jelasity, A. Montresor, and B. O., "Gossip-based aggregation in large dynamic networks," *ACM Transactions on Computer Systems*, vol. 23, no. 3, pp. 219–252, 2005.

[16] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "The implementation of lua 5.0," *Journal of Universal Computer Science*, vol. 11, no. 7, pp. 1159–1176, 2005.

[17] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 80–112, 1985.

[18] P. Grassé, "La reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la théorie de la stigmergie: Essai d'interprétation des termites constructeurs." *Insects Sociaux*, vol. 6, pp. 41–83, 1959.

[19] R. Nagpal, "A Catalog of Biologically-Inspired Primitives for Engineering Self-Organization," in *Engineering Self-Organising Systems*, G. Di Marzo Serugendo, A. Karageorgos, O. F. Rana, and F. Zambonelli, Eds. Springer Berlin Heidelberg, 2004, pp. 53–62.

[20] 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.

[21] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems," *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.

[22] J.-C. Baillie, "URBI: towards a universal robotic low-level programming language," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE Press, Piscataway, NJ, 2005, pp. 820–825.

[23] R. O'Grady, A. L. Christensen, and M. Dorigo, "SWARMORPH: Morphogenesis with Self-Assembling Robots," in *Morphogenetic Engineering: Toward Programmable Complex Systems*, R. Doursat, H. Sayama, and O. Michel, Eds. Springer Berlin Heidelberg, 2012, ch. 2, pp. 27–60.

[24] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys (CSUR)*, vol. 43, no. 3, p. Paper ID 19, 2011.

[25] R. Nagpal, "Programmable self-assembly using biologically-inspired multiagent control," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems part 1 - AAMAS '02*. New York, New York, USA: ACM Press, 2002, pp. 418–425.

[26] 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.

[27] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," *CoRR*, vol. abs/1202.5509, 2012. [Online]. Available: http://arxiv.org/abs/1202.5509

[28] 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.

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

[30] 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.

[31] 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.

[32] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87)*, vol. 21, no. 5. ACM New York, 1987, pp. 123–138.

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.