An Agent Framework for Agent Societies

Kyle Usbeck Jacob Beal *
BBN Technologies
Cambridge, MA, USA, 02138
{kusbeck,jakebeal}@bbn.com

Abstract

A key weakness of agent frameworks is the difficulty of specifying and controlling the global (emergent) behavior of the Multi-Agent System (MAS) in which they operate. The spatial computing language Proto, however, compiles descriptions of global behavior into local behaviors that interact to produce the specified emergent behavior. In this paper, we show how Proto can be used as a tool for construction of multi-agent systems, allowing the MAS designer to express the global behavior, while still creating a distributed solution. We compare and contrast Proto's functionality to that of existing agent frameworks, showing how Proto is a good candidate for the agent community's first agent framework for *societies* of agents.

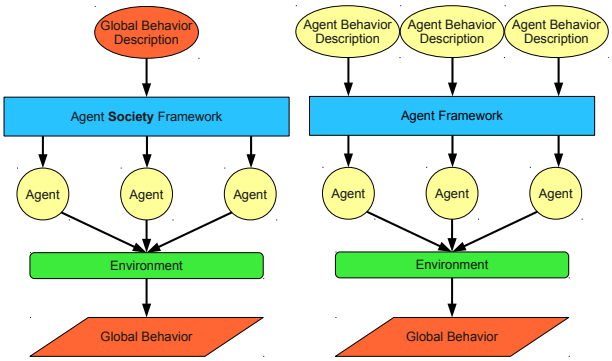
1. Introduction

Agents are typically modeled as autonomous processes with sensors and effectors. Agent frameworks (sometimes called agent architectures) are the toolkits that aid in their development. Often, agent-oriented programming is used to simplify the behavioral description of a complex system—decomposing a large problem into a multi-agent system. The global effect of the Multi-Agent System (MAS) is sometimes called an *emergent behavior* because it is not explicitly expressed, but "emerges" from the aggregate behaviors of the individual agents.

A key weakness of agent frameworks is their lack of ability to specify or control the global behavior of the agent system. Instead, agent frameworks provide tools for building single agents, in the hopes that their aggregate local behaviors will produce the desired global behavior.

Consequently, an agent society framework is needed for creating agents from a description of desired global behavior. We believe that the Proto [8] language and the collection of associated tools in the MIT Proto [21] reference implementation, although not designed for this purpose, are a good candidate for an initial agent society framework.

Proto goes beyond the scope of typical agent frameworks by allowing the programmer to specify the global behavior of a MAS in a simple and concise manner using a continuous space-time abstraction. This continuous space-time abstraction can be approximated by any network of devices with limited communication range, allowing the Proto compiler to automatically convert a global specification into a distributed program that is executed by individual devices. The local interactions of the devices then combine to produce an emergent behavior that necessarily approximates the global specification. Thus, for any society of locally-communicating agents distributed through a real or virtual space, we can use Proto to predictably design the behaviors of the agent society.



(a) Agent Society Framework (b) Traditional Agent Framework

Figure 1. Proto, an agent society framework, differs from traditional agent frameworks in that its input (a description of global behavior) is more closely related to the end goal (global multi-agent system behavior) than descriptions of individual agent behaviors, which are inputs to traditional agent frameworks.

Figure 1 shows how Proto differs from typical agent frameworks. Where traditional agent frameworks provide utilities for describing the behaviors of individual agents, Proto's input is a description of the desired global behavior of the society of agents. Such an *agent society framework* is a valuable tool for distributed systems because an MAS designer's goals are often global behaviors, and thus more readily expressed in terms of global behavior descriptions rather than individual agent descriptions.

The specific contributions of this paper are:

- A description of how agents may be represented in Proto,
- A mapping of Proto semantics and design patterns to functional concepts and architectural paradigms of agent frameworks,
- A review of relevant example implementations of the Proto agent society framework, and
- A list of open research challenges.

* Work sponsored by DARPA DSO under contract W91CRB-11-C-0052; the views and conclusions contained in this document are those of the authors and not DARPA or the U.S. Government.

SPLASH '11 October 22–27, Portland, USA.

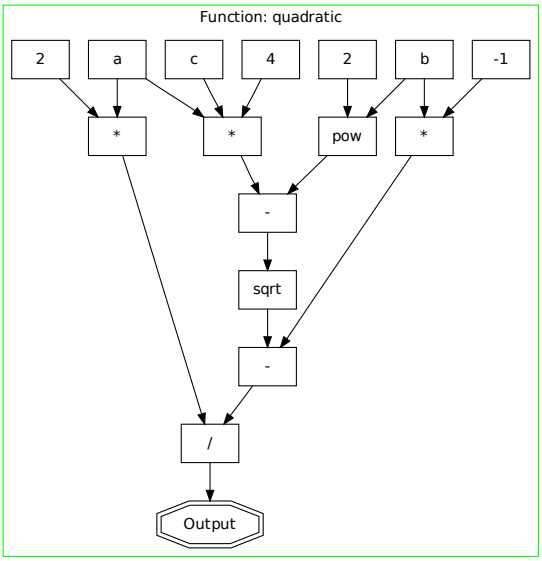


Figure 2. The Proto compiler produces, localizes, and optimizes dataflow graphs created from Proto programs. This figure shows an example dataflow graph for solving the quadratic formula.

2. Background and Approach

We begin with a brief review of Proto and the agent framework functional concepts and architectural paradigms upon which we map Proto.

2.1 Proto

MIT Proto[21], although often expressed as a single tool, has three separate components: a language, a compiler and a virtual machine. The Proto language[8] is a LISP-like pure-functional language (although Proto is not a LISP). Primitives in Proto are mathematical operations on fields, where a field is a function that maps every point in a space to some value. For example, if we take the surface of the Earth to be a space, then one field might map every location to its current temperature, while another field might map every location to its latitude and longitude. These fields may change over time as well, as temperature does.

The composition of these elements with computing operations produces a dataflow graph, such as the one in Figure 2, which depicts the dataflow graph of the quadratic formula:

```
(def quadratic (a b c)
  (/ (- (neg b)
        (sqrt (- (pow b 2)
                 (* 4 a c))))
     (* 2 a)))
```

These field-based computing operations can be categorized as having one of two purposes: *1)* creating patterns over space/time, and *2)* moving information across space. Proto programs, such as those shown in Section 4, are designed by combining these functional building-blocks to create the desired global behavior.

The MIT Proto compiler accepts the Proto language as input and processes it in three loosely-coupled stages. The first stage converts global descriptions of programs to functional operations on fields of values. Next the functional operations are transformed to a local execution on an amorphous medium [5], a computational model where every point in a manifold[1] is a computing device and each device has access to the recent past state of a nearby neighborhood. Finally, the continuous amorphous medium is approximated by the discrete network of the spatial computer.

The last component of MIT Proto is the ProtoKernel virtual machine[3]. This virtual machine has a core library and definitions for a set of platform-specific functions (e.g., commands to move a robot, report a debugging signal, broadcast a packet of data, etc.) that are meant to be implemented by platform developers. There are several existing implementations of the virtual machine, which will be discussed more in Section 5.

Other collective programming frameworks [12, 13, 19, 22] exist, but Proto's approach is unique. Butera's paintable computing [13] relies on the programmer for low-level network details, whereas Proto can make use of a space-time abstraction. Furthermore, paintable computing [13], TOTA [19], WaveScript [22], and Smart Messages [12] all adopt an agent-oriented programming paradigm that forces developers to concentrate on the individual device behaviors rather than their aggregate global behavior. Proto's advantages of specifying the global system behavior and utilization of the amorphous medium space-time abstraction make it a perfect tool for programming societies of agents.

2.2 Agent System Reference Model/Architecture

The Agent System Reference Model [27] (ASRM) defines functional concepts that are typical in agent frameworks. The Agent System Reference Architecture [23] (ASRA) builds on the ASRM by describing how these agents function within an agent community, or multi-agent system (MAS) and formalizes frameworks' architectural paradigms.

Both the ASRM and ASRA are created from static and runtime analyses of existing agent frameworks (e.g., JADE [31], Cougaar [17], and AGLOBE [16]). While neither document claims to define a complete set of agent functionality, both documents aim to capture the core set of common operations in major agent framework implementations.

The ASRM defines seven functional concepts for agent systems: 1) Agent Administration, 2) Directory Services, 3) Security and Survivability, 4) Messaging, 5) Mobility, 6) Conflict Management, and 7) Logging.

The ASRA elaborates on the ASRM by defining architectural paradigms for implementing the functional concepts within agent frameworks. The remainder of this section explains the functional concepts as defined in the ASRM and their architectural paradigms as defined in the ASRA.

2.2.1 Agent Administration

The ASRM defines the *Agent Administration* functional concept as:

> Agent administration functionality *a)* facilitates and enables supervisory command and control of agents and/or agent populations and *b)* allocates system resources to agents. Command and control involves instantiating agents, terminating agents, and inspecting agent state. Allocating system resources includes providing access control to CPUs, user interfaces, bandwidth resources, etc.

Architectural Paradigm: On-the-fly Administration On-the-fly agent administration allows agents to be added, removed, stopped, started, and configured while the agent system is running. It is not necessary to specify the number or types of agents *a priori*.

[1] A *manifold* is a geometric space that locally resembles Euclidean space, but globally may be more complex (i.e., the surface of a large sphere).

Architectural Paradigm: Pre-configured Administration Pre-configured agent administration requires the number and type of agents to be selected prior to running the agent system.

2.2.2 Directory Services

The ASRM defines the *Directory Services* functional concept as:

> Directory Services functionality facilitates and enables locating and accessing shared resources.

Architectural Paradigm: Subscription-based Services With subscription-based directory services, agents can query a central repository to discover services. Agents make subscriptions to particular services to register for any service changes (e.g., new agents offering the service, agents no longer offering the service).

2.2.3 Security and Survivability

The ASRM defines the *Security and Survivability* functional concept as:

> The purpose of security functionality is to prevent execution of undesirable actions by entities from either within or outside the agent system while at the same time allowing execution of desirable actions. The goal is for the system to be useful while remaining dependable in the face of malice, error or accident.

The ASRA does not currently contain architectural paradigms for Security and Survivability.

2.2.4 Messaging

The ASRM defines the *Messaging* functional concept as:

> Messaging functionality facilitates and enables information transfer among agents in the system.

Architectural Paradigm: Direct Messaging Direct messaging encodes inter-agent messages for transport and sends them in a one-to-one, one-to-many (multi-cast), or one-to-all (broadcast) style.

Architectural Paradigm: Shared-object Messaging Shared-object messaging is a blackboard-style [17] communication paradigm where messages are contributed to one or more message repositories—addressed to the agents for whom they are destined.

2.2.5 Mobility

The ASRM defines the *Mobility* functional concept as:

> Mobility functionality facilitates and enables migration of agents among framework instances typically, though not necessarily, on different hosts. The goal is for the system to utilize mobility to make the system more effective, efficient and robust.

Architectural Paradigm: Serialization Mobility The process of serialization mobility occurs when the framework stops the agent's execution on one platform, serializes the agent with its current state, sends the serialized agent to another platform (often using the *Messaging* functional concept described in Section 2.2.4), and starts execution of the agent on the new platform.

Architectural Paradigm: Shared-object (state) Mobility In shared-object mobility, the agent and its state are updated on all platforms. Along with the agent's state, the framework keeps track of the platform(s) on which the agent is executing. Each platform is responsible for monitoring which agents it should be running.

2.2.6 Conflict Management

The ASRM defines the *Conflict Management* functional concept as:

> Conflict management functionality facilitates and enables the management of interdependencies between agents activities and decisions. The goal is to avoid incoherent and incompatible activities, and system states in which resource contention or deadlock occur.

The ASRA does not currently contain architectural paradigms for Conflict Management.

2.2.7 Logging

The ASRM defines the *Logging* functional concept as:

> Logging functionality facilitates and enables information about events that occur during agent system execution to be retained for subsequent inspection. This includes but does not imply persistent long-term storage.

Architectural Paradigm: Direct Logging Direct logging writes to standard-out (e.g., C's `printf`, Java's `System.out`, etc.) or a language's built-in logging mechanism (e.g., `logging.Logger` from Python, `java.util.logging` from Java).

Architectural Paradigm: Indirect Logging Indirect logging writes to some external logging utility (i.e., `log4j` [1]).

2.3 Approach

We begin by mapping the general concept of "agent" into the Proto framework. Then, for each functional concept of the ASRM, we will discuss its implementation in Proto, mapping the behavior of Proto to an architectural paradigm from the ASRA where such exists. The purpose of this is to establish Proto as an agent framework for societies of agents. This also has the side benefit of contributing an additional example of agent framework analysis to the the ASRA, which is meant to be a "living document."

3. Mapping Agents to Proto

The term "agent" broadly refers to an autonomous entity with sensors and actuators. Due to the term's loose definition it is often overloaded across domains. In this section, we first examine the internals of Proto agents, then define the different notions of agents within Proto.

3.1 Agent Internals

Russell and Norvig [29] define the internal components of an agent as: sensors, effectors, and an agent program (including internal state) [2]. Sensors and effectors are handled in a device-specific manner in Proto, by means of an extensible plugin system. For instance, the code below shows sensor and effector declarations that are then used a program running with three different test sensors

[2] Note that the Russell and Norvig model is only one of a number of widely-used models of agent internals. As it is often possible to map these models to one another, we believe that Proto can be mapped to most other agent models as well.

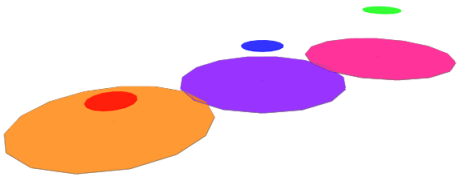


Figure 3. Three Proto nodes run in simulation with test sensors (highlighted color underneath the node) and `red`, `blue`, and `green` LED actuators at various heights.

(`sense` n) and three different actuators for enabling/disabling a device's colored LED at a given height (`red`, `blue`, and `green`).

```
;; Sensor/Effector declarations
(primitive sense (scalar) scalar)
(primitive red (scalar) scalar)
(primitive green (scalar) scalar)
(primitive blue (scalar) scalar)
;; Example program
(all
   (if (sense 1)    ;; if sensor-1 is enabled
      (red 5)          ;; turn on red LED at height 5
      (red 0))         ;; else, turn off red LED
   (if (sense 2)    ;; if sensor-2 is enabled
      (blue 10)        ;; turn on blue LED at height 10
      (blue 0))        ;; else, turn off blue LED
   (if (sense 3)    ;; if sensor-3 is enabled
      (green 15)       ;; turn on green LED at height 15
      (green 0)))      ;; else, turn off green LED
```

Figure 3 shows this code running in simulation on three nodes where the large circle indicates the test sensor and the LED is a smaller colored circle floating above the node at a specified height.

As defined by Russell and Norvig [29], an agent program "maps from a percept to an action" where percepts are combinations of sensors and internal state, and actions are combinations of effectors and state transitions. Proto expressions, such as the one shown above, can define such agent programs, but the details depend on how we draw the boundaries between agents. The remainder of this section discusses the two different notions of agent definitions and how they are represented in Proto.

3.2 Hardware vs. Software Agents

In hardware-oriented fields, an agent usually refers to a single device or machine. In contrast, software-oriented fields label autonomous processes as agents. Thus, many *software agents* may be running on a single *hardware agent*. On the other hand, multiple instances of a single software agent can execute on multiple hardware agents.

Proto can be mapped to notions of both software and hardware agents. In Proto, hardware agents are the devices that comprise the discrete approximation of the continuous space described by the amorphous medium abstraction. In other words, each device in the spatial computer network is a separate hardware agent. In Proto, each hardware agent runs the same program. For example, if the global behavior is `(red 1)`, then **all** the devices will enable their red LED. This example illustrates the case of a single software agent (a simple agent that enables a red LED) running on multiple hardware agents (devices).

Software agents in Proto, on the other hand, are more an interpretation of the structure of a program. We shall take a software agent to be a spatially-executed thread—a sub-program that runs on some set of devices in the spatial computer network. For example, using the `all` operator, an arbitrary number of separate functions are evaluated simultaneously, and each can represent a different agent.

```
(all
   ( AGENT ONE )
   ( AGENT TWO )
   ( AGENT THREE ))
```

In the above case, all three agents are run on every device in the network. If we wish to restrict the execution of an agent to a subset of devices, the `if` operator can be used.

```
(if ( CONDITION )
   ( AGENT ONE )
   ( AGENT TWO ))
```

In the case of restriction, agent one runs on any device where `condition` is true. Agent two runs only on the devices where `condition` is false.

These "software agent" examples illustrate the case of multiple agents running on the same physical device. In this paper, we compare and contrast the functional concepts provided by agent frameworks to the properties of both hardware and software agents in Proto.

4. Functional Concepts Supported in Proto

4.1 Agent Administration

In agent frameworks, the creation and destruction of agents is governed by the *Agent Administration* functional concept. There are a number of ways that Proto can "create" agents. Proto can be used to spawn new hardware agents using the `clone` operator and terminate agents using `die`. When a device is cloned, a new device is added to the spatial computer network and the global program is started on the new device.

For example, the following program clones an agent every second:

```
(if (= (mid) (timer)) ;; if current time equals the machine's ID #
   (clone (mid)) 0)   ;; clone the machine
```

However, varying the number of hardware agents in Proto has little effect on the global behavior unless the devices are constrained, physically or algorithmically, to maintain a particular density. When hardware devices are scattered sparsely in space, as in many swarm or sensor applications, changing the number of devices simply changes the quality of the global-to-local approximation, or the resolution of the space that the devices occupy.

Creating new software agents is accomplished in the global behavior description as described in Section 3.2. Proto allows sensors and effectors to be included as functions in the language, making agent programs simple and concise to read and write.

Proto's Agent Administration is an example of **On-the-fly Administration**. The global behavior description only contains information about how the collective agents behave; but administration attributes, such as the number, structure, and distribution of agents are determined at run-time.

4.2 Directory Services

In many agent frameworks, directory services are implemented by providing naming and query-matching services, similar to

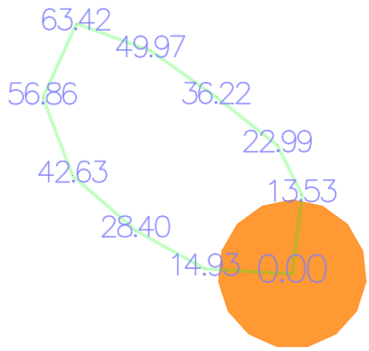


Figure 4. Proto provides *content-based addressability* to find the nearest service provider. In this example, each node calculates and displays their Euclidean distance to the service provider (orange).

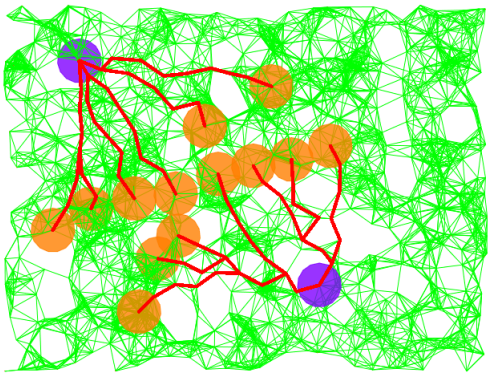


Figure 5. Proto's directory services simplify task replication as shown in this example where the client nodes (orange) draw their shortest path to any service provider (purple).

UDDI [14]. Proto provides *content-based addressability* as a way searching for agents that provide a specific service.

For example, the following program, shown in Figure 4 indicates that the devices should calculate their Euclidean distance to the closest device whose boolean sensor (sensor number one) is activated:

```
(distance-to (sense 1))   ;; display distance to sensor
```

The significance of this statement is that `(sense 1)` returns a mapping of values (in this case true if the sensor is enabled, false otherwise) to devices. Thus, Proto's content-based addressability has a direct correlation to the typical mapping of services to their URI.

Furthermore, content-based addressability simplifies the replication of tasks. In the following example, shown in Figure 5, multiple `(sense 1)` listeners are each connected to their closest `(sense 2)` service.

```
(connect (sense 1) (sense 2)) ;; display shortest path from all
                              ;; sensor-1 to the closest sensor-2
```

In a slightly more complex example, consider the most common architectural paradigm of directory services in agent frameworks: **Subscription-based Services**, or publish-subscribe (pubsub). Proto can be used to emulate pubsub functionality with a function such as the one below, where a set of *subscribers* listen for a *stream* from a *service*.

```
(def pubsub (service subscribers stream)
  ;; if subscriber is subscribed to service
  (if (fold + 0 (connect subscribers service))
    ;; send the subscriber any updates via stream
    (broadcast service stream 0)))
```

The ASRA lacks a decentralized approach to Directory Services, such as the one used by Proto. This is an example of an area where Proto might help to guide the future development of the ASRA.

4.3 Security and Survivability

Agent frameworks need to protect the agent platform from undesirable actions. This notion of *security* could mean preventing harmful code (e.g., from a migrated agent) from executing. Similarly, *survivablilty* describes the agent system's ability to remain "dependable in the face of malice, error or accident" [27].

Security is an open area for research in Proto, while survivability is one of Proto's greatest strengths. Because survivability is often implicit in the specification of the global behavior of the system, Proto programs have a high-degree of survivability naturally.

[6, 9, 10] talk about self-repairing functions, such as gradient-based distance calculations (e.g., the one used in `distance-to`). These functions have the property that, if a change occurs during computation, the function will correct its values in a predictable short amount of time. Furthermore, the feed-forward composition of such self-repairing functions also results in self-repairing composite functions.

For example, the following program can be used to compute all the nodes along the shortest path between two devices (source and destination):

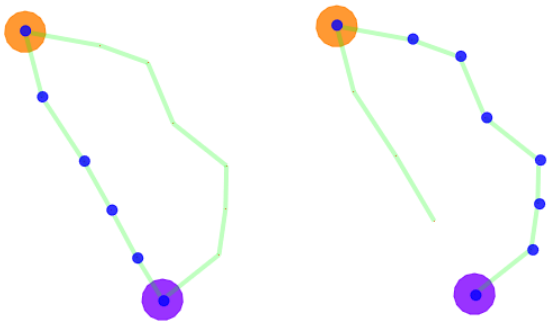
```
(def shortest-path (source destination)
 (letfed
         ;; di distance from src to dest from executing node
         ((di (+ (distance-to source)
                 (distance-to destination)))
         ;; min-di is the shortest path distance
          (min-di (min-hood
                   (broadcast destination di))))
  ;; if executing node is on the shortest path
  (if (and (not (= min-di (inf)))
           (= min-di di))
   (blue 1)      ;; turn on blue LED
   (blue 0))))   ;; else, turn off blue LED
```

The program executes at every time-step, re-calculating the nodes along the shortest path and enabling their blue LED. If, after the shortest path is computed, a change affects the shortest path (i.e., node movement causes a network disconnection), the shortest path is recomputed and updated automatically. Figure 6 shows Proto agents automatically re-computing the shortest path (in blue) after a disruption in the network occurs.

4.4 Messaging

Standard agent frameworks tend to provide an API for communication between agents. The purpose of the API is to abstract the agent communication protocol from the agent programmer. Proto devices take this abstraction to the next level with *implied communication*.

With implied communication, devices share their state with their neighbors after every round of program execution. This allows the programmer to easily write programs whose information is distributed across many devices. In Proto, implied communication is contained in neighborhood functions (e.g., `int-hood`), which are



(a) Before Disruption (b) After Disruption

Figure 6. Proto's implicit survivability is illustrated by the shortest-path program that re-calculates the nodes along the shortest path (in blue) after a disruption occurs in the network.

used for implementing the following common pattern in distributed algorithms:

- Gather data from neighbors,
- Do some computation separately on each neighbor's data, and
- Return a value combining the results from the neighbors.

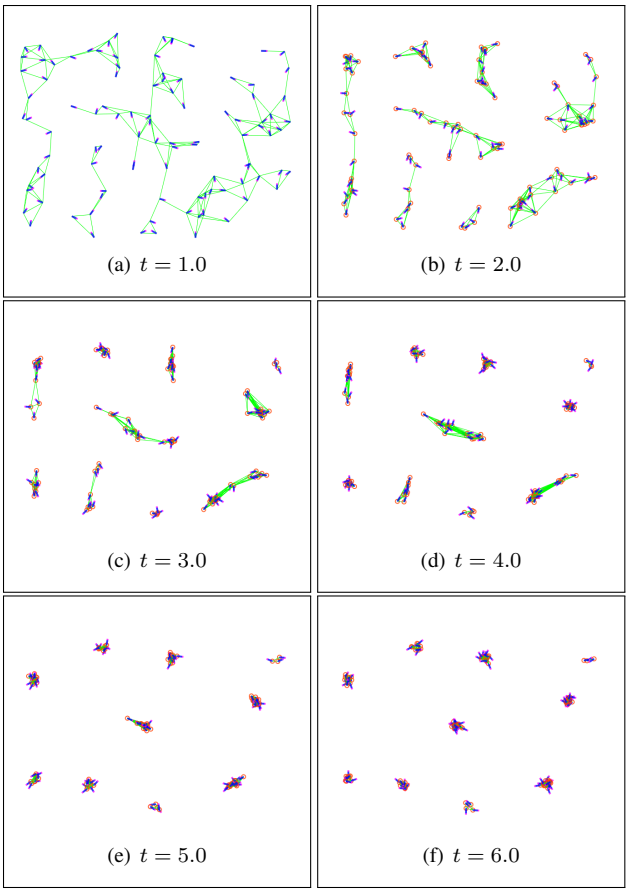
The following program, depicted in Figure 7, makes use of implied messaging and device mobility (discussed in Section 4.5) as a way of clustering devices:

```
(mov                    ;; move the device
   (normalize           ;; normalize the vector
      (int-hood         ;; integrate over each neighbor's vector
         (nbr-vec))))   ;; return distance-vector to each neighbor
```

In this program, `nbr-vec` is used as a way of acquiring the vector-distance to each neighbor, and then `int-hood` is used to compute the integral of the vectors over the neighborhood. These two combine to imply that devices must send messages exchanging localization information.

In traditional agent frameworks, there are many barriers to emulate this messaging functionality. First, individual (local) agent behaviors are defined. Next, a protocol is established for communicating between devices. Of course, if multiple messages from different agents are received concurrently, then the messaging protocol must be sufficiently expressive to represent each individual evaluation. By implicitly sharing state between neighboring devices in the spatial computer, Proto saves the programmer from implementing the individual agent behaviors and messaging protocol.

Software agents in Proto also use a form of implied communication. Communication between operator evaluations are accomplished using *fields*—mappings of values to devices. For instance, the following function definition, `nav-grad`, shown in Figure 8,



(a) $t = 1.0$ (b) $t = 2.0$ (c) $t = 3.0$ (d) $t = 4.0$ (e) $t = 5.0$ (f) $t = 6.0$

Figure 7. Implicit messaging (and device mobility) allows Proto devices to share their locations to create clusters without defining an inter-agent communication protocol.

passes messages from navigation agents (green) in order to direct moving agents (blue) toward a source device (orange).

```
(def share-distance-to (is-calculating source)
  (let ((base (if is-calculating        ;; base names the
                  (distance-to source)  ;; distance to source
                  (inf))))              ;; or infinity
    (green (< base (inf)))  ;; enable green LED if immobile
    (mux is-calculating     ;; if mobile
         base               ;; move toward source
         ;; else, pass along navigation directions to neighbors
         (min-hood (+ (nbr-range) (nbr base))))))

(def nav-grad (is-mover source)
       ;; g is a distance-based gradient to source
  (let ((g (grad (share-distance-to
                      ;; only compute g on non-movers
                      (not is-mover) source))))
      ;; if I'm a mover agent and the gradient is non-zero
      (mux (and is-mover (> (len g) 0))
         ;; follow the gradient to the source
         (normalize g)
         ;; all other agents remain stationary
         (tup 0 0))))
```

In the navigation agents, the variable `g` names a self-healing gradient [6] based on the distance to the source device. That gradient is maintained based on navigation information passed between agents in the `share-distance-to` function.

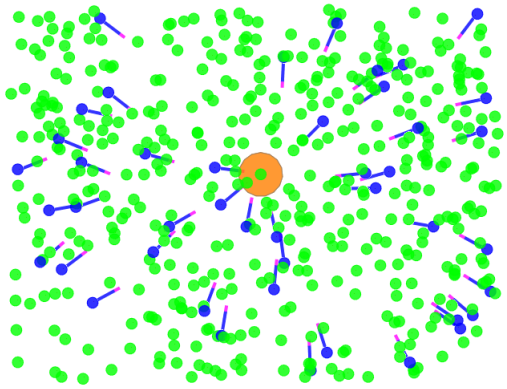


Figure 8. Proto software agents use language scoping rules to pass navigational messages through immobile agents (green), that direct "mover" agents (blue) toward a source device (orange).

Communication between software agents is governed by the scope of the variables between the agents. Similar to LISP, variable scope in Proto is lexical (static), however `let` blocks allow a variable to have dynamic scope. Thus, the messages between Proto software agents are passed through shared memory, often made available by the hardware agent's implied communication.

Communication occurs between the software agents in the form of fields, where a field is a mapping of values to devices in the network. Dataflow graphs, such as the one diagramed in Figure 2 depict these fields as arrows between operators.

Proto's implied communication is a form of **Direct Messaging** as described in the ASRA because, after every computation step, the agent's state is serialized and broadcast to all its neighbors.

4.5 Mobility

Mobility in hardware agents is accomplished by moving the physical devices. A special actuator, `mov`, accepts a 3-vector as input and moves the device in the direction of the vector, at the speed described by the vector's magnitude. The movement actuator is described in more detail in [4].

Modifying the execution space of a software agent over time *mobilizes* that agent. In other words, mobility means the same program executes on different machines at different times. [7] describes some of the inherent complexities with identifying and manipulating processes with spatial extent.

For example, the code below, shown running in Figure 9, migrates an agent across a network once a sensor is triggered. To do so, this code relies on *(a)* a sensor being activated and *(b)* implicit messaging (see Section 4.4) to spread information across the network. The information contains a flag that indicates if the sensor has been triggered—indicating that the agent should be executing. Figure 9 shows how the agent (which enables a red LED) propagates itself throughout the network over time. Furthermore, when a node moves to connect a previously disconnected network segment (in Figure 9(f)), the mobile agent showcases its survivablilty by migrating to the newly connected nodes.

```
;; enable the red LED on devices with a running agent
(red (rep running 0
       ;; if an agent is running
       (if (= (mid) running)
          ;; look for sensor-1 on any neighbors
          (+ (any-hood (nbr (sense 1))) running)
          ;; else, mobilize the agent to neighbors
          (max-hood (nbr running)))))
```

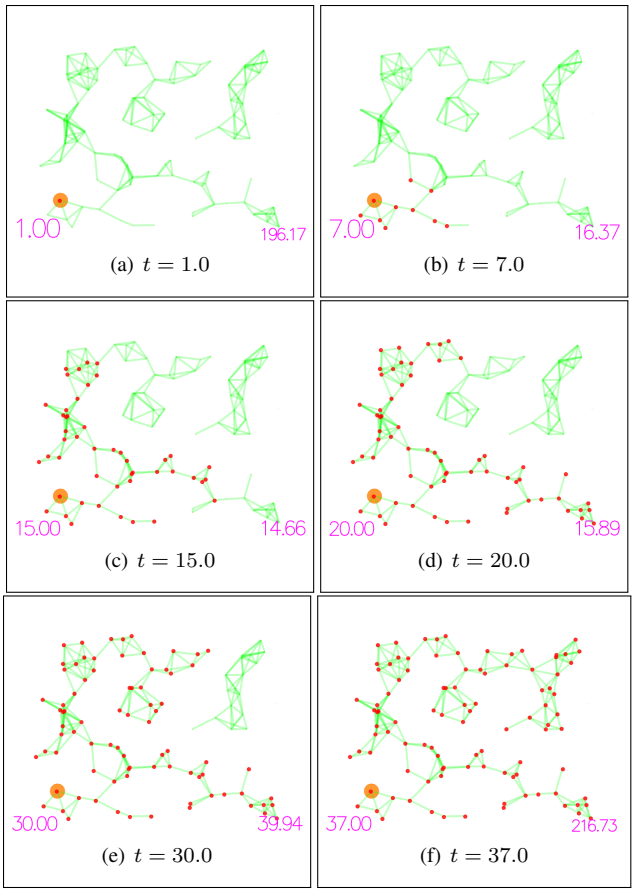


Figure 9. Mobility in Proto is controlled by restricting agent execution to different spaces (nodes) at different times. In this example, the mobile agent migrates to network neighbors and turns an LED red. Note that in Figure 9(f) a node moves to connect the two previously disjoint networks.

Proto's notion of mobility is a form of **Shared-object Mobility** because the agent behavior and its local state exist on each Proto device. Mobility is controlled by simply changing the operational space—or branch of execution—of the device. In our example, both agents and all their necessary state exist on all of the devices. An agent migration occurs when the switch changes and execution ends on one node and begins on another[3].

4.6 Conflict Management

The lexical scoping of the Proto language prevents some conflicts from occurring. Other than the lexical scoping rules, there are currently no built-in facilities for managing conflicts in Proto. However the language does not preclude conflict mediation. For example, the `elect` operator is a self-stabilizing symmetry-breaking function, used for cooperative election of regional leaders in a network. The following code, shown executing in Figure 10, illustrates

[3] Migration actually occurs between sets of nodes, not just single nodes.

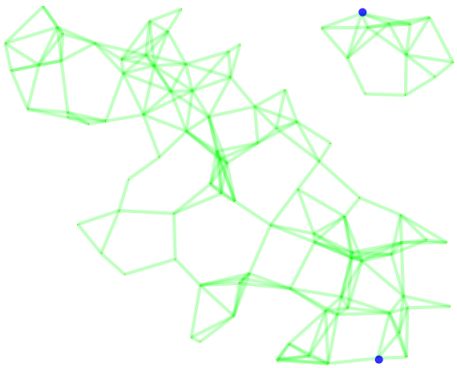


Figure 10. Proto makes use of self-stabilizing symmetry-breaking functions for conflict management as shown by the cooperatively elected leaders (blue) in each network.

`elect` working in a partitioned network where blue nodes are the elected leaders.

```
(def elect (radius)
  ;; pick a random value (0-1)
  (let ((id (once (rnd 0 1))))
    ;; leader has the lowest value
    (= id (rep minid id
               ;; compute distance to the leader
               (let ((dist (distance-to (= id minid))))
                 (mux (> dist radius)
                      ;; leader returns its value
                      id
                      ;; establish an area around the leader
                      (mux (<= dist (* 0.25 radius))
                           ;; inside the area is the leader's value
                           ;; (the lowest of the group)
                           (min-hood (nbr minid))
                           ;; outside that area is infinity
                           (inf))))))))
(blue (elect 400)) ;; enable the elected leaders' blue LEDs
```

In the `elect` operator, every device randomly choses a value from (0–1). The device with the lowest value is designated as the leader. A small area around the leader assumes the leader's value, and everything outside that area uses the value `infinity`, up until it is far enough away that a new leader can be elected for another region. When two areas overlap, only the lower-value area retains its elected leader. Thus, the set of leaders self-stabilizes to produce a pattern where all leaders are separated by at least half the specified region `radius` and no device is more than `radius` from some leader. Of course, these devices are cooperating during the election process. Operation with non-cooperative agents is an open research area for Proto.

4.7 Logging

Distributed logging is a current area of research [2, 28]. One particular method that has been explored in Proto is having each device maintain its own list of log messages. Processing the log messages then becomes similar to making a query on a distributed database [11, 25]. Logging is an area of future work for Proto.

5. Framework Implementations based on Proto

This section illustrates the breadth of applicability of Proto by briefly describing several implementations of the Proto framework for different platforms. Typically, the framework is applied to a platform by creating a variant implementation of the ProtoKernel virtual machine that is integrated with that platform's systems for communication, sensing, and actuation.

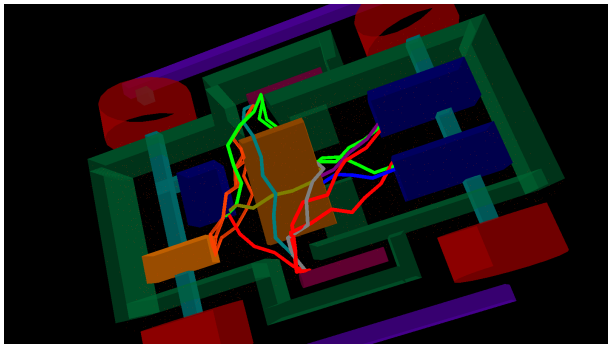


Figure 11. ProtoSim is a robust, extensible simulator for Proto agents. In this example, Proto is being used to semi-automatically wire the internal components of a robot.

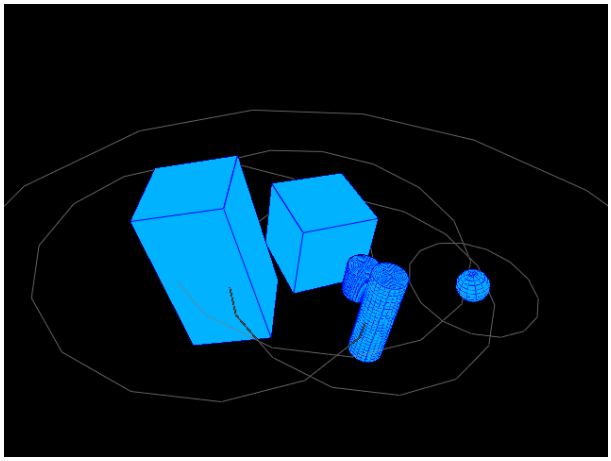
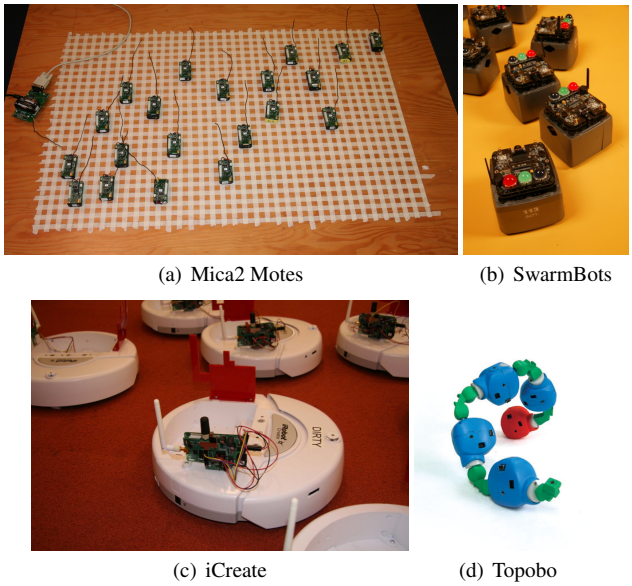


Figure 12. A ProtoSim plugin allows Proto agents to receive collision-detection and realistic physics feedback from the Open Dynamics Engine (ODE).

5.1 ProtoSim

ProtoSim is the reference implementation of networked execution of Proto code, bundled with MIT Proto and tied to the reference implementation of the ProtoKernel virtual machine. The simulator takes Proto code, invokes the reference compiler to produce a ProtoKernel binary, and executes it on a 2D or 3D distribution of simulated devices, each running an instance of the reference ProtoKernel virtual machine. This is visualized in 3D using OpenGL [24] for display and to allow the user to manipulate the running devices, but may also be run in a non-interactive mode without visualization for batch simulation. Figure 11 shows ProtoSim running a Proto program to semi-automatically wire the internal components of a robot.

ProtoSim includes a plugin system for adding sensors and effectors, adjusting communication models, or a number of other such modifications. For example, the Open Dynamics Engine (ODE) [30] was integrated with ProtoSim to provide feedback to Proto agents for collision-detection and a realistic physics. Figure 12 shows devices with different body-types interacting within ProtoSim.



(a) Mica2 Motes (b) SwarmBots

(c) iCreate (d) Topobo

Figure 13. Proto has been applied to hardware platforms for sensor networks (a), swarm robotics (b,c), and modular robotics (d). (Photo credit: (b) James McLurkin and Swaine Photography, (c) Nikolaus Correll, (d) Hayes Raffle and Amanda Parkes)

5.2 Hardware Platforms

Proto has also been implemented on a number of hardware platforms (Figure 13). One family of these are sensor network platforms, such as Mica2 Motes [18]. These are typically based on a small AVR processors, with highly constrained processing and memory (e.g., 16MHz 8-bit process and 4K of RAM on the Mica2), for which ProtoKernel is well-suited. Proto has also been implemented on swarm robotics platforms, such as McLurkin and iRobot's swarm robots [20] and the iRobot iCreate [15]. On these platforms, resources tend to be more plentiful, as the energy constraints are typically dominated by movement, rather than computation, sensing, or communication. Proto has also been implemented on the Topobo [26] modular robotics platform, where the user assembles a structure of mixed passive and active physical linkages and wired USB network connectors.

On such hardware platforms, Proto is typically hosted as a component within the individual agent's operating system. For example, on the Mica2, Proto runs on top of TinyOS, and on the iCreate is a thread in OpenWRT Linux. Other agent services, such as localization or fine-grained motor control, are connected with Proto as sensors and actuators.

6. Open Research Challenges

The ASRM and ASRA do not state the minimum requirements for an agent framework, but instead explain the potential feature-sets of such systems through functional concepts and architectural paradigms. Proto provides many such feature sets, but there are a number of open research challenges in providing some of the functional concepts.

First, there is little support for managing conflicts between disparate groups of Proto agents. This is largely due to the fact that Proto has been used exclusively for cooperative agent societies. Research in non-cooperative Proto agents is thus an open research area.

Likewise, while Proto nodes are resilient to small errors in signals and large topology changes [4, 6], they have no inherent ability to resist malicious information. Security is therefore a major open research area for Proto.

Finally, Proto lacks a viable system for distributed runtime logging (although ProtoSim does implement a shared-output logging system). One potential approach identified for this area is to treat logging as a distributed database problem, perhaps drawing on existing data-collection paradigms developed by the sensor networks community.

7. Conclusion

Traditional agent frameworks fall short in that they provide functionality only for defining individual agent behaviors rather than defining global MAS behaviors. In this paper, we first map notions of agents to Proto's concepts of agents. Next, using the functional concepts and architectural paradigms laid out in the ASRM [27] and ASRA [23], we define the functional concepts supported by Proto—both the language and the compiler. Finally, we discuss some of the relevant virtual machine implementations of Proto and the open research challenges for developing Proto as a fully featured agent framework. Despite these challenges, we argue that Proto's simple and elegant implementations of agent society interactions make it a viable candidate to become the first agent framework for agent societies.

References

[1] Apache. log4j logging services, July 2011. URL `http://logging.apache.org/log4j/`.

[2] J. Avarias, J. López, C. Maureira, H. Sommer, and G. Chiozzi. Introducing high performance distributed logging service for acs. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7740, page 114, 2010.

[3] J. Bachrach and J. Beal. Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017, MIT, March 2007.

[4] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous space programs for robotic swarms. *Neural Computing and Applications*, 19 (6):825–847, 2010.

[5] J. Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, volume 3566 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin, September 2004.

[6] J. Beal. Flexible self-healing gradients. In *ACM Symposium on Applied Computing*, pages 1197–1201, New York, NY, USA, March 2009. ACM.

[7] J. Beal. Dynamically defined processes for spatial computers. In *Spatial Computing Workshop*, 2009.

[8] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.

[9] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In *ACM Symposium on Applied Computing*, New York, NY, USA, March 2008. ACM.

[10] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-stabilization for gradients. In *Distributed Computing in Sensor Systems (DCOSS) 2009*, June 2009.

[11] D. Bell. *Distributed Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1992. ISBN 0201544008.

[12] C. Borcea, D. Iyer, P. Kang, A. Saxena, and L. Iftode. Spatial programming using smart messages: Design and implementation. In *IEEE International Conference on Distributed Computing Systems*, 2004.

[13] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, MA, USA, 2002.

[14] L. Clement, A. Hately, C. von Riegen, and T. Rogers. Universal description, discovery and integration. Techincal Specification, Feb. 2005. URL `http://uddi.xml.org`.

[15] N. Correll, J. Bachrach, D. Vickery, and D. Rus. Ad-hoc wireless network coverage with networked robots that cannot localize. In *IEEE International Conference on Robotics and Automation, Kobe, Japan*. IEEE Press, May 2009.

[16] Czech Technical Institute Agent Technology Center. Aglobe, Oct. 2011. URL `http://agents.felk.cvut.cz/aglobe/`.

[17] A. Helsinger, M. Thome, and T. Wright. Cougaar: a scalable, distributed multi-agent architecture. In *IEEE Systems, Man and Cybernetics, 2004*, volume 2, pages 1910–1917. IEEE, 2004.

[18] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[19] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: the TOTA approach. *ACM Transactions on Software Engineering and Methodology*, 2008.

[20] J. McLurkin. Stupid robot tricks: A behavior-based distributed algorithm library for programming swarms of robots. Master's thesis, MIT, 2004.

[21] MIT Proto. MIT Proto. software available at `http://proto.bbn.com/`, Retrieved November 22, 2010.

[22] R. Newton, L. Girod, M. Craig, S. Madden, and G. Morrisett. Wavescript: A case-study in applying a distributed stream-processing language. Technical Report Technical Report MIT-CSAIL-TR-2008-005, MIT CSAIL, January 2008.

[23] D. N. Nguyen, K. Usbeck, W. M. Mongan, C. T. Cannon, R. N. Lass, J. Salvage, and W. C. Regli. A methodology for developing an agent systems reference architecture. In *11th International Workshop on Agent-oriented Software Engineering*, Toronto, ON, May 2010.

[24] OpenGL. The industry standard for high performance graphics, July 2011. URL `http://www.opengl.org/`.

[25] M. Ozsu, M. Ăzsu, and P. Valduriez. *Principles of distributed database systems*. Springer, 2011.

[26] H. Raffle, A. Parkes, and H. Ishii. Topobo: a constructive assembly system with kinetic memory. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 647–654. ACM, 2004.

[27] W. C. Regli, I. Mayk, C. J. Dugan, J. B. Kopena, R. N. Lass, P. J. Modi, W. M. Mongan, J. K. Salvage, and E. A. Sultanik. Development and specification of a reference model for agent-based systems. *Trans. Sys. Man Cyber Part C*, 39:572–596, September 2009. ISSN 1094-6977. doi: 10.1109/TSMCC.2009.2020507. URL `http://portal.acm.org/citation.cfm?id=1656816.1656823`.

[28] T. Ropars and C. Morin. Improving message logging protocols scalability through distributed event logging. *Euro-Par 2010-Parallel Processing*, pages 511–522, 2010.

[29] S. Russell, P. Norvig, J. Canny, J. Malik, and D. Edwards. *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, NJ, 1995.

[30] R. Smith. Open dynamics engine, Aug. 2011. URL `http://www.ode.org/`.

[31] Telecom Italia Lab. JADE — Java Agent DEvelopment framework, Oct. 2011. URL `http://jade.tilab.com/`.