2011 Fifth IEEE International Conference on Self-Adaptive and Self-Organizing Systems

# Modeling Self-* Systems using Chemically-Inspired Composable Patterns

Akla-Esso Tchao, Matteo Risoldi, Giovanna Di Marzo Serugendo
*CUI - Université de Genève*
*Carouge, Switzerland*
{*akla-esso.tchao, matteo.risoldi, giovanna.dimarzo*}*@unige.ch*

***Abstract*—The behaviour of self-* systems is complex to model from an algorithmic point of view. Designing and specifying self-* systems implies a great amount of work that can be sensibly reduced if models can be reused and composed in a modular way. This article discusses a chemically-inspired architecture and formalisms that facilitate the creation of modular, reusable models based on behavioural patterns inspired by behaviours found in nature. The architecture is based on chemical-like laws ruling the evolution of the system. We show the reuse of general behavioural patterns using three concrete examples of self-* systems from different domains.**

***Keywords*-self-* systems; modular; reuse; model**

## I. Introduction

Complex self-* pervasive systems often have non-trivial, emergent behaviours which are not explicitly present in their specifications, but appear as a result of interactions. These behaviours can sometimes be seen as a composition of simpler behaviours (or "patterns") that are frequently found in the domain. This can be a recursive composition: an algorithm may be composed of patterns, which in their turn are composed of simpler patterns. For example, a routing algorithm may rely on a pattern for building gradients, itself composed of a spreading and an aggregation pattern. Thus gradient diffusion, spreading and aggregation can be seen as a simple recurring patterns at different levels. The design pattern paradigm, generally applied in other fields of computer science like software design, software architecture and human-computer interaction, can bring great benefits to the design of self-* pervasive systems.

Traditionally, these systems are coded algorithmically. This has some disadvantages, among others the fact that algorithms are generally very specific to the system being coded, limiting reusability and making their composition difficult especially for what concerns concurrency. Since a few years, approaches based on the original chemical abstract machine paradigm try to solve this problem by modeling systems using chemically-inspired formalisms, where "reactions" between system components are specified by rewriting rules. This is a step forward towards composition of simple behavioural patterns: if a behaviour is specified as a rewriting system, then it is possible to compose different behaviours via rewriting system composition [1] (modulo appropriate verification for keeping concurrency and convergence properties).

The work presented in this paper is part of an EU-funded project named SAPERE. The SAPERE approach uses chemically-inspired models to specify system behaviours, with rules called *eco-laws* governing the evolution of ecosystems of *live semantic annotations* (LSAs) that reify agents, services and data. Behaviours are expressed in terms of composition of simple behavioural patterns found in nature. The contribution of SAPERE is a model for pervasive systems that is scalable, modular, decentralised and easy to verify. In this first phase of this project, we want to show that the SAPERE model is expressive and general enough to reproduce existing self-organising decentralised algorithms.

The goals of this article are: *(i)* showing how existing self-organising decentralised algorithms are modeled in SAPERE; *(ii)* illustrating a few of the simple behavioural patterns that have been classified until now using the SAPERE model; and *(iii)* showing how patterns can be composed to obtain more complex behaviours. We will start with a brief description of the SAPERE model, followed by a discussion of a few behavioural patterns. We will then illustrate pattern reuse and composition with three different examples (crowd steering, a routing protocol and ant foraging).

## II. The SAPERE Model

The SAPERE approach is inspired by chemical mechanisms [2]. This section gives a general overview of the approach; a complete description is available in [3].

SAPERE takes its primary inspiration from natural ecosystems. Unlike the many proposals that adopt the term "ecosystem" simply as a mean to characterise the complexity and dynamics of modern ICT systems, SAPERE exploits nature-inspired mechanisms for actually ruling the overall system dynamics.

SAPERE considers modelling and architecting a pervasive service environment as a non-layered spatial substrate, laid above the actual network infrastructure. The substrate embeds the basic "laws of nature" or "eco-laws" that rule the activities of the system. It represents the ground on which the components of a pervasive service ecosystem interact and combine with each other (in respect of the eco-laws and based on their spatial relationships), so as to serve their own individual needs as well as the sustainability of the overall ecology. Users can access the ecology in a decentralised way

978-0-7695-4542-4/11 $26.00 © 2011 IEEE
DOI 10.1109/SASO.2011.22

to use and consume data and services, and they can also act as prosumers by injecting new data or service components.

SAPERE adopts a common modelling and treatment of services, data, and devices. All "entities" living in a system (services, data, digital/network resources in general, including devices) have an associated semantic representation called *Live Semantic Annotation* (LSA). This is a very basic ingredient for enabling dynamic unsupervised interactions between components. LSAs are evolving, live entities, tightly associated to the component they describe, and capable of reflecting its current situation and context. They act as actual observable interfaces of resources, as well as the basis for enforcing semantic and self-aware forms of dynamic interactions (both for service aggregation/composition and for data/knowledge management). The dynamics of the ecosystem are driven by *eco-laws*, defining the basic policies to rule a sort of virtual "chemical reactions" among the LSAs of the various individuals of the ecology. LSAs are like chemical reagents in an ecology in which interactions and composition occur via reactions, i.e., semantic pattern-matching, between LSAs. Such reactions can contribute to establish bonds between entities (e.g., relating similar services with each other to produce a distributed service, or mining related data items) as well as to produce new components (e.g., a composite service orchestrating the execution of atomic service components or a high-level knowledge concept derived from the aggregation of raw data items). The overall self-aware holistic adaptivity of the system is ensured by the fact that any change in the system (as well as any change in its components, as reflected by dynamic changes in its LSA) will reflect in the firing of new chemical reactions, possibly leading to the establishment of new bonds and/or in the breaking of existing bonds.

### A. Summary of eco-law language

A language has been developed to express LSAs and eco-laws and is presented in [4]. For readability purposes, we will only summarise it briefly and informally.

LSAs are tuples of typed values $\langle v_1, \ldots, v_n \rangle$, where $v_i$ can be of an arbitrarily defined type, e.g, numbers, strings or structured values. For example, LSA $\langle \texttt{SEN1}, \texttt{temp}, 30 \rangle$ may be the LSA representing a temperature reading of 30 degrees by a sensor SEN1.

Eco-laws are reactions on LSA templates[1]. A template is any LSA, which may or may not have variables in the tuple. An LSA $L$ is said to match a template $P$ if a variable substitution $S$ exists s.t. $S(P) = L$. We conventionally use `typetext` when writing LSA elements which are concrete values, and $italics$ when writing variables. For example, template $\langle id, \texttt{temp}, 30 \rangle$ both matches LSAs $\langle \texttt{SEN1}, \texttt{temp}, 30 \rangle$ and $\langle \texttt{SEN2}, \texttt{temp}, 30 \rangle$. An eco-law is hence of the kind $P_1, \ldots, P_n \overset{r}{\mapsto} P'_1, \ldots, P'_m$, where: *(i)* the left-hand side (reagents) specifies templates that match LSAs $L_1, \ldots, L_n$, which are to be extracted from the LSA-space; *(ii)* the right-hand side (products) specifies the LSAs which are accordingly to be inserted back in the LSA-space (after applying substitutions found when extracting reagents, as in standard logic-based rule approaches); and *(iii)* rate $r$ is a numerical positive value indicating the average frequency at which the eco-law is to be fired. An eco-law might not have an associated rate; in this case, its firing can happen as soon as the left-hand templates match.

In addition to local reactions, eco-laws can involve reading/modifying LSAs in neighbouring nodes. For this, a special syntax has been devised. Creating an LSA $L$ in an unspecified neighbouring node (to be chosen non-deterministically at runtime) is indicated as $+\langle L \rangle$ in the right-hand side of an eco-law. The same notation can be found in the left-hand side of an eco-law, meaning that $L$ is being read from the neighbour instead. If the eco-law needs to create/modify an LSA in a specific node $N$ instead of choosing a random neighbour, we use the syntax $N\langle L \rangle$. If instead an eco-law needs to create/modify an LSA in *all* its neighbours *at once*, the $bcast\langle L \rangle$ syntax is used.

## III. Behavioural Patterns with Eco-laws

A number of behavioural patterns have been classified and studied to-date. These range from elementary patterns (e.g., aggregation, spreading, evaporation), to middle level patterns (e.g., gradient, gossip) to higher-level behaviours (e.g., chemotaxis, morphogenesis, quorum sensing). In previous works [5], [6] we described these patterns from a general point of view, using an abstract notation (useful for describing the theory, but not for creating models). The eco-law language, described above, is a more concrete notation we use to describe and implement these patterns in the SAPERE model. The examples we will see in the next section concentrate on showing reuse of four patterns: spreading, aggregation, evaporation and gradient. We will now briefly describe each of these patterns and give their corresponding instantiation in the SAPERE model using eco-laws.

### A. Patterns and abstraction

There are several advantages to taking a modular, pattern-based approach to specifying pervasive systems. First and foremost, the reuse of previously defined patterns facilitates the design of new systems, and reduces development time and cost. Also, modularity supports evolution and maintainability, and facilitates ensuring correctness and dependability of a modular pervasive system with software verification techniques like, e.g., model checking [7].

From a more specific system modeling point of view, patterns enable us to describe a system with different levels of abstraction. We can identify a set of *core* low-level

[1] In other works [4] these have been called *LSA patterns*; however, in order not to generate confusion with the behavioural patterns, we will use here the term *template*.

patterns (described in the following), that express generic, simple, nature-inspired behaviours not specific to any particular domain or application. Depending on the way they are integrated, they can express different high level behaviours. Moreover, these models abstract away certain aspects; for example, in the AODV routing model illustrated in this article, the sender and target could be either physically or socially connected, however the patterns forming the model remain the same.

### B. The spreading pattern

The Spreading Pattern is a low-level pattern for information diffusion/dissemination. It progressively sends information over the system using direct communication among agents. The spreading of information in a system allows the agents to increment the global knowledge of the system by using only local interactions.

In the SAPERE model, spreading an LSA $L$ is achieved with a simple eco-law:

$$L \xmapsto{R_{spr}} L, +L \tag{1}$$

This means that if $L$ is present (precondition: the LSA is written in the left-hand side), it can be consumed, and in its place $L$ itself is created locally ($L$) and in a neighbour node ($+L$) chosen non-deterministically. Variants of this eco-law exist with respect to the choice of the neighbour node, namely using the $N\langle L\rangle$ and $bcast\langle L\rangle$ notations explained in Section II-A. Note that $L$ appears in the local node on both sides of the law; this means it is "kept" locally as well as being spread. However, it is possible to consider a variant of spreading where the LSAs are just moving from one node to the other (i.e., are consumed from one node and are created in the other). An example of this pattern could be an LSA representing an agent moving from one node to another:

$$\langle agent\rangle \xmapsto{R_{spr}} +\langle agent\rangle \tag{2}$$

### C. The aggregation pattern

The Aggregation Pattern reduces the amount of information in the system and assesses meaningful information. It is useful for example when repeated application of the above spreading pattern creates multiple copies of the same LSA in a node. Aggregation can consist in applying mathematical functions on LSA values, or on eliminating redundant LSAs. An example eco-law for the latter case could be:

$$L, L' \xmapsto{R_{agg}} L \tag{3}$$

where $L$ and $L'$ are two equivalent LSA; this eco-law eliminates redundancy by only keeping one. For an example where a mathematical function is applied, consider a node having two LSAs, injected by different agents at different times, indicating quantities $q$ and $q2$ of a chemical $chem$. We want the two to be aggregated in one LSA with the maximum of the two values. The eco-law would be:

$$\langle chem, q\rangle, \langle chem, q2\rangle \xmapsto{R_{agg}} \langle chem, max(q, q2)\rangle \tag{4}$$

Here we apply the $max$ operator to the quantities $q$ and $q2$, keeping the highest.

### D. The evaporation pattern

Evaporation is a pattern that helps dealing with dynamic environments where the information used by agents can become outdated. Evaporation is generally modeled by eco-laws consuming LSAs with a certain rate:

$$L \xmapsto{R_{evp}} L' \tag{5}$$

where $L'$ is generally an LSA with a smaller value. The $R_{evp}$ rate might be tied to the age of the LSA (obtained, e.g., from its timestamp). For example, a chemical that evaporates might be described by the following eco-law:

$$\langle chem, q\rangle \xmapsto{R_{evp}} \langle chem, q * EvFactor\rangle \tag{6}$$

The evaporation function can be arbitrarily complex; here we simply multiplied the old value by a variable $EvFactor \in [0 - 1)$. Another variant of evaporation may completely remove an LSA. For example, the chemical might completely disappear:

$$\langle chem, q\rangle \xmapsto{R_{evp}} \tag{7}$$

### E. The gradient pattern

The Gradient Pattern is an extension of the Spreading Pattern where the information is propagated in such a way that it provides an additional information about the source's distance. Either a distance attribute is added to the information (the gradient increases with distance), or the value of the information is modified such that it reflects its concentration (the gradient decreases with distance). Additionally, the Gradient Pattern uses the Aggregation Pattern to merge different gradient values created by different agents.

In the SAPERE model, a gradient G initiates from a *source LSA* $S$ that is injected in a node:

$$S \xmapsto{R_{grInit}} S, G_0 \tag{8}$$

where $G_0$ is the LSA with the initial value of the gradient. The gradient then spreads, using aggregation to eliminate redundant LSAs:

$$G_i \xmapsto{R_{grSpr}} G_i, +G_j \tag{9}$$

$$G_i, G_j \xmapsto{R_{grAgg}} G_i \tag{10}$$

where $G_j$ has a higher distance (resp. lower concentration) value than $G_i$. Since different types of LSAs are involved in gradients, for the sake of clarity it can be useful to mark source LSAs with a value `source` contained in the LSA. A source LSA will often have a form similar to $\langle \texttt{source}, gradient, max, annealing \rangle$, where $gradient$ is the name for the gradient, $max$ is the maximum value the gradient LSAs can have, and $annealing$ is a parameter for the gradient often used to tune the gradient evaporation[2]. Gradient LSAs are instead marked as such: $\langle \texttt{grad}, gradient, value, max, annealing \rangle$. This is not to say this is a general form for gradient LSAs, as more or less parameters may be needed depending on the model; however, we found this form to be recurrently useful in the models we created.

To make an example, a gradient $g$ (expressing distance in increments of 1) initiates through the following eco-law:

$$\begin{array}{l} \langle \texttt{source}, g, max, A \rangle \xmapsto{R_{grInit}} \\ \langle \texttt{source}, g, max, A \rangle, \langle \texttt{grad}, g, \texttt{0}, max, A \rangle \end{array} \tag{11}$$

where `0` is the gradient value at the source. The gradient LSA is then diffused by the following couple of eco-laws (a spreading, and an aggregation keeping the LSA with the shortest distance):

$$\begin{array}{l} \langle \texttt{grad}, g, val, max, A \rangle \xmapsto{R_{grSpr}} \\ \langle \texttt{grad}, g, val, max, A \rangle, +\langle \texttt{grad}, g, val{+}\texttt{1}, max, A \rangle \end{array} \tag{12}$$

$$\begin{array}{l} \langle \texttt{grad}, g, val, max, A \rangle, \langle \texttt{grad}, g, val{+}i, max, A \rangle \\ \xmapsto{R_{grAgg}} \langle \texttt{grad}, g, val, max, A \rangle \end{array} \tag{13}$$

## IV. Evaluation Criteria for the Examples

The following sections will show how to model three different examples by modular composition of the above patterns. To assess the quality of the model obtained with eco-laws, we used the criteria we had already applied in our previous work [8], comparing the model either to the requirements (in the crowd steering example) or to existing standard algorithms (for the AODV and ants foraging examples). The criteria are:

- *Convergence* — if the model reaches the desired goal;
- *Speed of convergence* — how quickly the model converges;
- *Stability* — if the behaviour of system agents appears to focus on goals;
- *Scalability* — how (and how much) convergence and speed are influenced by the number of LSAs;

We chose not to concentrate too much on performance at this stage, since this depends on implementation aspects. However, performance is comparable between classic algorithms and eco-law models.

The three examples have been chosen to show three different contexts of application of the SAPERE model. The crowd steering example shows how the eco-law instantiating the patterns are put together to create a new model from scratch. The AODV example shows how the SAPERE model can be used to express a classic algorithm with static agents. The ant foraging example, finally, also shows modeling a classic algorithm with SAPERE, but with dynamically moving agents.

## V. Example 1: Crowd Steering

Our first example of reuse of patterns is taken from [4], where it is discussed in detail. It is a crowd evacuation application, in which a fire breaks out in a museum and people have to evacuate based on indications received by their PDAs. This example was built from scratch for [4] (i.e., we did not build on an existing algorithm), and coded directly with eco-laws.

The surface of the exposition is covered by sensors, arranged in a grid, able to sense fire, detect the presence of people, interact with other sensors in their proximity as well as with PDAs that visitors carry with them. When a fire breaks out, PDAs (by interaction with sensors) must show the direction towards an exit, along a safe path. This is achieved with three gradients: the *exit gradient* (expressing distance from the exit), the *fire gradient* (expressing distance from the fire); and the *crowding gradient* (expressing distance from possible crowds that might block escape paths). These three gradients are diffused using the gradient (laws 14, 15 and 16) and evaporation (law 17) patterns discussed above:

$$\begin{array}{l} \langle \texttt{source}, G, M, A \rangle \xmapsto{R_{init}} \\ \langle \texttt{source}, G, M, A \rangle, \langle \texttt{grad}, G, \texttt{0}, M, A \rangle \end{array} \tag{14}$$

$$\begin{array}{l} \langle \texttt{grad}, G, V, M, A \rangle \xmapsto{R_s} \\ \langle \texttt{grad}, G, V, M, A \rangle, +\langle \texttt{grad}, G, \texttt{min}(V{+}\texttt{1}, M), M, A \rangle \end{array} \tag{15}$$

$$\begin{array}{l} \langle \texttt{grad}, G, V, M, A \rangle, \langle \texttt{grad}, G, W, M, A \rangle \mapsto \\ \langle \texttt{grad}, G, \texttt{min}(V, W), M, A \rangle \end{array} \tag{16}$$

$$\langle \texttt{grad}, G, V, M, A \rangle \xmapsto{R_{ann}(A)} \langle \texttt{grad}, G, V{+}\texttt{1}, M, A \rangle \tag{17}$$

Note that these eco-laws are parameterised on the gradient's name (the $G$ variable). This lets a single set of eco-laws rule the behaviour of all gradients (a further step towards efficient reuse, thanks to parameterisation).

[2]For example, an eco-law defining the evaporation of a gradient could have a rate tied to the annealing value, e.g.:

$$\langle g, value, max, A \rangle \xmapsto{R_{evp}(A)} \langle g, value * EvRate, max, A \rangle$$

These gradients are used to compute each node's *attractiveness* value, an aggregation of gradient values expressing the relative safety of a node as an escape path:

$$\begin{array}{l}\langle \texttt{grad}, \texttt{exit}, E, Me, Ae\rangle, \langle \texttt{grad}, \texttt{fire}, F, Mf, Af\rangle, \\ \langle \texttt{info}, \texttt{crowd}, CR, TS\rangle \xmapsto{R_{att}} \\ \langle \texttt{grad}, \texttt{exit}, E, Me, Ae\rangle, \langle \texttt{grad}, \texttt{fire}, F, Mf, Af\rangle, \\ \langle \texttt{info}, \texttt{crowd}, CR, TS\rangle, \langle \texttt{info}, \texttt{attr}, f(E, F, CR), \#T\rangle \end{array} \tag{18}$$

where $f$ is a calculation based on the gradient's values ($f \propto \frac{F}{E \times CR}$). The attractiveness value is finally used for choosing escape paths.

### A. Evaluation

This example was made from scratch, and not as a re-modeling of an existing algorithm. It was simulated with an ad-hoc simulator called *Alchemist* [4], that models execution of eco-laws as CTMC transitions, where the eco-law rate is interpreted as a Markovian rate. We also introduced node failures to assess resilience. With respect to our evaluation criteria, it evaluates as follows:

- *Convergence* — the goal is all persons in the exposition leaving through the exits; this was reached with different experimental conditions (number and position of people and fires, node failures), up to nodes failing 80% of the time (convergence was not guaranteed beyond this limit);
- *Speed of convergence* — the metric is how quickly all people leave the exposition; this was found to be influenced by the weight of the crowding gradient (affecting jam formation) and by failing nodes. We found an optimum crowding weight for which the number of ticks to convergence remained in an acceptable interval up to 80% node failure.
- *Stability* — the persons should always focus on heading towards the exits; this was also observed, within the 80% failure limits, using gradients as a guide;
- *Scalability* — convergence is not influenced by the number of persons; the model converged even with large crowds. Speed of convergence instead was influenced in a considerable way due to crowding of the exit and passageways. This was actually foreseen and reflects real-world crowd dynamics.

## VI. Example 2: AODV Routing

### A. Simulation environment

This and the next example were implemented using a software framework for agent-based simulation called Repast [9]. It provides an integrated library of classes for creating, running, displaying and collecting data from an agent-based simulation. At its heart, Repast behaves as a discrete event simulator whose quantum unit of time is known as a tick. The tick exists only as a hook on which the execution of events can be hung, ordering the execution of the events relative to each other. Therefore, a Repast simulation is primarily a collection of agents of any type and a model that sets up and controls the execution of these agents' behaviours according to a schedule. This schedule not only controls the execution of agent behaviours, but also actions within the model itself, such as updating the display, recording data, and so forth.

*1) Scheduling:* In Repast we implemented each eco-law as a specialized method. Each one is triggered according to the following scheduling. At each tick $t$ of the simulator, all eco-laws are evaluated. Each executable eco-law at that tick $t$ is internally re-scheduled in Repast to execute at tick $t+\delta t$, with $\delta t$ smaller than the advancing step size of the global simulator tick. An eco-law is executable if the left hand side LSAs of the eco-law matches the LSAs in the space. All internally scheduled eco-laws in Repast for tick $t+\delta t$, are triggered concurrently in a random order. The Repast engine ensures that each eco-law is picked up once and executed concurrently along with the others.

The actual firing of an eco-law is additionally submitted to the probability given by its rate $r$. For instance an executable eco-law with probability rate of 50% will have 50% of chance to actually execute. In addition, when the execution rate is satisfied, the eco-law reagents are re-evaluated before execution. This to ensure that concurrent eco-laws (e.g. acting and modifying the same set of input LSA) take into account the actual new state of the LSAs before triggering their own actions. Algorithm 1 summarizes this scheduling.

**Algorithm 1** Scheduling of eco-laws in Repast

```
1:  EcolawList := List of all eco-laws
2:  EcolawRateMap := Map holding the execution rate associated to each eco-law
    in EcolawList
3:  ToExecuteList := List holding executable eco-laws          {Initially empty}
4:  foreach ecolaw in EcolawList
5:  if (ecolaw is executable) then
6:      ToExecuteList := ToExecuteList ∪ ecolaw                {Add the executable
        eco-law to the list}
7:  end foreach
8:  if (ToExecuteList != ∅) then
9:      do concurrently foreach ecolaw in ToExecuteList        {Repast internal}
10:     p := random uniform number between 0 and 1
11:     r = EcolawRateMap[ecolaw]   {get the rate associated to ecolaw}
12:     if (p ≤ r) then
13:         if (ecolaw is executable) then
14:             trigger ⟨ Eco-law ecolaw ⟩   {Executes eco-law with a rate r}
15:     end do
```

### B. AODV overview

Mobile ad-hoc network (MANET) is a self-configuring infrastructureless network of mobile nodes. In MANET, there is no centralised node to coordinate the flow of messages to each node in the network.

The Ad-hoc On Demand Distance Vector Routing (AODV) [10], [11] is a routing algorithm for the operation of such ad-hoc networks. AODV allows mobile nodes to obtain routes for new destinations quickly, and does not require

them to maintain routes to destinations that are not in active communication.

Route Requests (RREQs), Route Replies (RREPs), and Route Errors (RERRs) are the message types defined by AODV.

A node (called *source)* disseminates a RREQ when it determines that it needs a route to a destination and does not have one available in its cache. To do this, the source generates the RREQ message and broadcasts it to its neighbours. The RREQ contains the following fields:

$$\{sourceAddress, sourceSequenceNumber, broadcast\_id, destinationAddress, destinationSequenceNumber, hopCount\} \quad (19)$$

The node's neighbours then forward the request to their own neighbours, and so on until either the destination or an intermediate node with a "fresh enough" route to the destination is located. The pair $\langle sourceAddress, broadcast\_id\rangle$ uniquely identifies a RREQ. The *broadcast_id* is incremented for new RREQs. RREQs from same node with the same *broadcast_id* will not be broadcast more than once. The source sequence number is used to maintain freshness of information about the reverse route to the source, and the destination sequence number specifies how fresh a route to the destination must be before it can be accepted by the source. As the RREQ travels from a source to the destination, it automatically sets up the reverse path from all nodes back to the source. To set up a reverse path, a node records the address of the neighbour from which it received the first copy of the RREQ.

When a destination node or a node that has an active route to the destination receives the RREQ, it generates a RREP message. The RREP contains the following fields:

$$\{sourceAddress, destinationAddress, destinationSequenceNumber, hopCount, lifetime\} \quad (20)$$

Once created, the RREP is unicast to the next hop toward the originator of the RREQ. As the RREP travels back to the node that generated the RREQ message, the hop count field is incremented by one at each hop. A node receiving an RREP propagates the first RREP for a given source node towards that source. If it receives further RREPs to that source, it updates its routing information and propagates the RREP only if the RREP contains either a greater destination sequence number than the previous RREP, or the same destination sequence number with a smaller hopcount. If the generator of the RREP is the destination itself, it increments its own sequence number by one if the sequence number in the RREQ message is equal to its value. Otherwise, the destination does not change its sequence number before generating the RREP message. On receiving the first RREP, the source can begin data transmission. If the source discovers a better route, the routing information can be updated.

If either the destination or some intermediate node moves or fails, a RERR message is sent to the affected source nodes. The node upstream of the break point initiates the RERR by listing each of the destinations that are now unreachable because of the broken link. It sends the RERR to its precursor nodes. Each precursor nodes marks the route to the destination as invalid, and sends the RERR further to its precursor nodes. When the source receives the RERR, it initiates the route discovery again if the route is still necessary. The RERR contains the following fields:

$$\{numberOfUnreachableDestinations(destCount), unreachableDestinationAddress\} \quad (21)$$

### C. AODV with Eco-Laws and LSA

The AODV algorithm has been modeled with eco-laws. For simplification reasons, in order not to obtain a model too large to be easily readable in the article, we did not model the error mechanism and the RREQ refreshing using the source sequence number. Note that, however, both these mechanisms can also be modeled easily using the same patterns described herein. We also simplified in assuming that all eco-law rates are 1 (thus, they will always execute when possible).

There are three types of LSAs in this model. One is for the messages:

$$\langle dest, \texttt{MSG}, sentRequest\rangle$$

where $dest$ is the id of the destination node, `MSG` marks the LSA type, and $sentRequest$ indicates if a route request has been already sent for this message.

The second LSA type is for route requests (RREQ):

$$\langle reqID, \texttt{RREQ}, source, dest, senderID, processed\rangle$$

where $reqID$ is the request id (corresponding to the *broadcast_id* in the description 19 of RREQ), `RREQ` is the LSA type, $source$ is the id of the source, $dest$ is the id of the destination of the message which triggered the request, $senderID$ is the id of the sender of the RREQ, and $processed$ is a boolean indicating if the request has already been processed (i.e., if it has already been re-broadcast).

The third LSA type is for route replies (RREP):

$$\langle \texttt{RREP}, source, dest, senderID, DSN, hopcount, expiry\rangle$$

where `RREP` is the LSA type, $source$ is the ID of the source node, $dest$ is the ID of the destination node, $senderID$ is the ID of the node who created the RREP, $DSN$ is the destination sequence number, $hopcount$ tracks the hop number the reply went through, and $expiry$ is a timestamp marking the time at which the RREP should evaporate.

Eco-laws for the AODV algorithm are as follows. First, if there is a message LSA in a node, and no RREQ was yet created for it, it triggers the creation of an RREQ LSA:

$$\langle t, \texttt{MSG}, \texttt{0}\rangle \mapsto \langle t, \texttt{MSG}, \texttt{1}\rangle, \langle \texttt{nxtReqID}(), \texttt{RREQ}, \texttt{nId}(), t, \texttt{nId}(), 0\rangle \quad (22)$$

where: nxtReqID() returns the next available request ID; nId() is the id of the current node (which is the source); and $t$ is the destination node taken from the message LSA. Note that the message is tagged with a 1 in its $sentRequest$ element.

If a node has a non-processed request, it is tagged as processed and broadcast to the neighbours:

$$\langle reqID, \texttt{RREQ}, s, t, sen, 0\rangle \mapsto \langle reqID, \texttt{RREQ}, s, t, sen, 1\rangle, bcast\langle reqID, \texttt{RREQ}, s, t, \texttt{nId}(), 0\rangle \quad (23)$$

Note that the broadcast request will have the current node ID (returned by nId()) as a sender.

If a node receives an RREQ LSA that it had already processed earlier, the new request is ignored:

$$\langle reqID, \texttt{RREQ}, s, t, sen, 0\rangle, \langle reqID, \texttt{RREQ}, s, t, sen, 1\rangle \mapsto \langle reqID, \texttt{RREQ}, s, t, sen, 1\rangle \quad (24)$$

The attentive reader will have recognised that eco-laws 22, 23 and 24 conform to the gradient pattern.

When the request reaches the destination (coming from another node $sen$), a route reply is sent to $sen$:

$$\langle reqID, \texttt{RREQ}, s, \texttt{nId}(), sen, processed\rangle \mapsto sen\langle \texttt{RREP}, s, \texttt{nId}(), \texttt{nId}(), \texttt{nxtDsn}(), 1, \texttt{time}()+\#LIFE\rangle \quad (25)$$

where time() is the current time in the system; $\#LIFE$ is the life duration for RREPs (a system constant); time()+$\#LIFE$ is thus the time at which the RREP must expire (i.e., evaporate); nxtDsn() returns the next available destination sequence number.

A node that receives an RREP to an RREQ it had received earlier, forwards the reply to the neighbour node from which the RREQ was received:

$$\begin{array}{l}\langle \texttt{RREP}, s, t, sen, dsn, hop, exp\rangle,\\ \langle reqID, \texttt{RREQ}, s, t, sen2, 1\rangle \mapsto\\ \langle \texttt{RREP}, s, t, sen, dsn, hop, exp\rangle,\\ \langle reqID, \texttt{RREQ}, s, t, sen2, 1\rangle,\\ sen2\langle \texttt{RREP}, s, t, \texttt{nId}(), dsn, hop{+}1, exp\rangle\end{array} \quad (26)$$

If a node receives two RREPs for the same pair source-destination, it keeps the one with the highest destination sequence number (i.e., the most recent):

$$\begin{array}{l}\langle \texttt{RREP}, s, t, sen, dsn, hop, exp\rangle,\\ \langle \texttt{RREP}, s, t, sen2, (dsn2 : dsn2{>}dsn), hop2, exp2\rangle \mapsto\\ \langle \texttt{RREP}, s, t, sen2, dsn2, hop2, exp2\rangle\end{array} \quad (27)$$

If the two RREPs have the same destination sequence number, the RREP with the smallest hop count is kept:

$$\begin{array}{l}\langle \texttt{RREP}, s, t, sen, dsn, hop, exp\rangle,\\ \langle \texttt{RREP}, s, t, sen2, dsn, hop{+}n, exp2\rangle \mapsto\\ \langle \texttt{RREP}, s, t, sen, dsn, hop, exp\rangle\end{array} \quad (28)$$

Again, eco-laws 25, 26, 27 and 28 conform to the gradient pattern.

If a node contains a message to send and an RREP indicating the path to the destination, the node sends the message to a neighbour towards the destination — i.e., the neighbour from which the RREP was received (spreading pattern):

$$\begin{array}{l}\langle t, \texttt{MSG}, sent\rangle, \langle \texttt{RREP}, s, t, sen, dsn, hop, exp\rangle \mapsto\\ sen\langle t, \texttt{MSG}, sent\rangle, \langle \texttt{RREP}, s, t, sen, dsn, hop, exp\rangle\end{array} \quad (29)$$

where the message and reply are matched via the $t$ parameter (i.e., the target identifier).

If an RREP is past its expiry time (i.e., if the expiry time is smaller or equal to the current time), the RREP evaporates (evaporation pattern):

$$\langle \texttt{RREP}, s, t, sen, dsn, hop, (exp : exp \leq \texttt{time}())\rangle \mapsto \quad (30)$$

Note that the RREP is completely removed from the node (it does not appear in the right-hand side).

Listing 1 gives an example of code implementing an eco-law (this is for eco-law 22).

Listing 1. Code for eco-law 22 that generates RREQs

```
  /**
   * @param xMsg the LSA triggering the
   * generation of the RREQ message
   */
private void msgToRreqLaw(LSAs.Message xMsg){
    rreqMsg.reqId = nxtReqID();
    rreqMsg.msgType = "RREQ";
    rreqMsg.sourceId = this.nId();
    rreqMsg.targetId = xMsg.targetId;
    rreqMsg.senderId = this.nId();
    rreqMsg.processed = false;
    //...
 }
```

### D. Evaluation

In order to evaluate the original AODV algorithm and the eco-law version, we implemented both of them in Repast. The original version, called simply *AODV*, follows the description given in Section VI-B. The eco-law version, called *AODV with eco-laws*, implements the eco-laws given in Section VI-C, using the scheduling in Algorithm 1. Figure 1 show a screenshot of AODV with eco-laws in Repast.

In terms of our evaluation criteria, AODV and AODV with eco-laws gave the following results:

- *Convergence* — both AODV and AODV with eco-laws established a route from the source to the destination for any couple ⟨Source, Destination⟩ chosen. In both cases, the established route is always the one with the smallest hop count among possible routes in the RREP messages.
- *Speed of convergence* — the speed of convergence in Repast for AODV and AODV with eco-laws was measured by the number of ticks needed to establish a

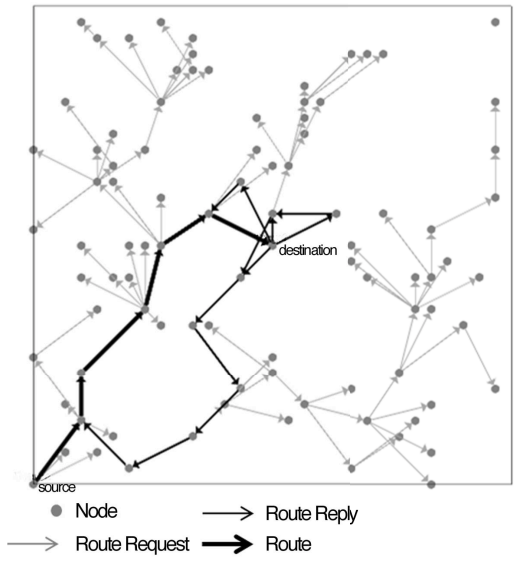


Figure 1. AODV with eco-laws

route from the source to the destination. Using the same topology and a fixed pair ⟨Source, Destination⟩, we had the following results for 10 runs of the simulation:

| Nodes | Ticks: AODV | Ticks: AODV eco-law |
|---|---|---|
| 50 | 18 | 18 |
| 100 | 18 | 18 |
| 150 | 18 | 18 |
| 200 | 18 | 18 |

AODV and AODV with eco-laws take in average the same time to converge. There is no delay induced by the fact that eco-laws were used. The number of ticks doesn't change because we use the same pair ⟨Source, Destination⟩. In fact, when a route with the smallest number of hops is found within 18 ticks, adding more nodes in the same topology doesn't change the number of ticks needed to establish it.

- *Stability* — for all simulated pair ⟨Source, Destination⟩ and size of the network, if a route exists, AODV and AODV with eco-laws found it. The established route contains the smallest number of hops among possible routes in the RREP messages received by the source.
- *Scalability* — variating the number of nodes between 50 and 200 did not affect in a significant way convergence and speed of convergence in either version of AODV.

## VII. Example 3: Ant Foraging

In this example, a colony of ants forages for food [12]. An area has an ant nest and three sources of food. The nest diffuses a *scent*, that indicates the distance of every location from the nest. Ants leaving the nest initially wander at random until they find food. When a piece of food is found, the ant carries it back to the nest, dropping a chemical as it moves (thus creating a chemical trail). When other ants sense the chemical, they follow the chemical towards the food. As more ants carry food to the nest, they reinforce the chemical trail. At any time, while searching for food or following a trail, ants may introduce random changes in their trajectory (*wiggling*). This example is part of NetLogo's standard library.

### A. Ant Foraging with Eco-laws

For the implementation of ant foraging using eco-laws, we chose to use the same parameters as the NetLogo version, and we used the scheduling described in Algorithm 1. The foraging algorithm was translated into the following set of eco-laws. First, a gradient with the distance from the nest is established with the gradient pattern (recalling the gradient structure in Section III-E, assuming a gradient source LSA for the nest is injected at the nest location by the system setup):

$$\langle \texttt{source}, \texttt{nest}, M, A\rangle \mapsto \\ \langle \texttt{source}, \texttt{nest}, M, A\rangle, \langle \texttt{grad}, \texttt{nest}, 0, M, A\rangle \tag{31}$$

$$\langle \texttt{grad}, \texttt{nest}, V, M, A\rangle \mapsto \\ \langle \texttt{grad}, \texttt{nest}, V, M, A\rangle, bcast\langle \texttt{grad}, \texttt{nest}, V{+}1, M, A\rangle \tag{32}$$

$$\langle \texttt{grad}, \texttt{nest}, V, M, A\rangle, \langle \texttt{grad}, \texttt{nest}, V{+}N, M, A\rangle \mapsto \\ \langle \texttt{grad}, \texttt{nest}, V, M, A\rangle \tag{33}$$

where $M$ and $A$ are the maximum and annealing parameters.

The gradient pattern is also used to model how the chemical scent left by the ants while they return to the nest after finding food spreads to the neighbouring locations, creating a "halo" effect:

$$\langle \texttt{grad}, \texttt{chem}, V, M, A\rangle \mapsto \langle \texttt{grad}, \texttt{chem}, V, M, A\rangle, \\ bcast\langle \texttt{grad}, \texttt{chem}, (V{*}\#DRATE), M, A\rangle \tag{34}$$

$$\langle \texttt{grad}, \texttt{chem}, V, M, A\rangle, \langle \texttt{grad}, \texttt{chem}, V{+}N, M, A\rangle \mapsto \\ \langle \texttt{grad}, \texttt{chem}, V{+}N, M, A\rangle \tag{35}$$

where $\#DRATE \in [0{-}1)$ is a system constant tuning how far the chemical spreads. This gradient is initialized at value of 0 in all locations.

The chemical evaporates with time, using the evaporation pattern (the evaporation speed $\#ERATE$ is configurable by the user in the GUI):

$$\langle \texttt{grad}, \texttt{chem}, V, M, A\rangle \mapsto \langle \texttt{grad}, \texttt{chem}, (V{*}\#ERATE), M, A\rangle \tag{36}$$

Ants are represented by LSAs $\langle \texttt{ant}, hasFood\rangle$, where $hasFood$ indicates if the ant is carrying food. Ants looking for food will eventually find themselves at the margin of a chemical trail, where the chemical concentration is between 0.5 and 2.0; at that point they will start following it (*spreading pattern*):

$$\begin{array}{l}\langle \texttt{ant}, \texttt{false}\rangle, \langle \texttt{grad}, \texttt{chem}, (V{:}V\ in\ [0.5{-}2.0])\rangle, M, A\rangle, \\ +\langle \texttt{grad}, \texttt{chem}, V{+}N, M, A\rangle \mapsto \\ +\langle \texttt{ant}, \texttt{false}\rangle, \langle \texttt{grad}, \texttt{chem}, V, M, A\rangle, \\ +\langle \texttt{grad}, \texttt{chem}, V{+}N, M, A\rangle\end{array} \quad (37)$$

From time to time ants will just change their direction at random, a behaviour called *wiggling* (*spreading pattern*):

$$\langle \texttt{ant}, \texttt{false}\rangle \mapsto +\langle \texttt{ant}, \texttt{false}\rangle \quad (38)$$

Note that wiggling can happen at any time, whether an ant is inside or outside the chemical trail (i.e., they can wiggle inside the trail); eco-law 37 will keep them inside the track, as reaching its margin will stimulate them to follow it again.

An ant finding food will change its $hasFood$ element to `true` and take a unit of food (*aggregation pattern*):

$$\langle \texttt{ant}, \texttt{false}\rangle, \langle \texttt{food}, V{+}1\rangle \mapsto \langle \texttt{ant}, \texttt{true}\rangle, \langle \texttt{food}, V\rangle \quad (39)$$

After finding food ants will start following the `nest` gradient to return to the nest, leaving a droplet of chemical (with a value of 60) in the path (*chemotaxis pattern*, described in [5]. Note that the neighbour node, choosen non-deterministically by the $+$ operator, is the same in the left and right member:

$$\begin{array}{l}\langle \texttt{ant}, \texttt{true}\rangle, \langle \texttt{grad}, \texttt{nest}, V{+}1, M, A\rangle, \\ +\langle \texttt{grad}, \texttt{nest}, V, M, A\rangle \mapsto \\ +\langle \texttt{ant}, \texttt{true}\rangle, \langle \texttt{grad}, \texttt{nest}, V{+}1, M, A\rangle, \\ +\langle \texttt{grad}, \texttt{nest}, V, M, A\rangle, \langle \texttt{chem}, 60\rangle\end{array} \quad (40)$$

The droplets are combined with the pre-existing chemical trail to refresh it (*aggregation pattern*), up to a maximum $M$ ($M$=200 in our simulations):

$$\begin{array}{l}\langle \texttt{chem}, Q\rangle, \langle \texttt{grad}, \texttt{chem}, C, M, A\rangle \mapsto \\ \langle \texttt{grad}, \texttt{chem}, \texttt{min}(C{+}Q, M), M, A\rangle\end{array} \quad (41)$$

Ants can of course also wiggle during the return trip:

$$\langle \texttt{ant}, \texttt{true}\rangle \mapsto +\langle \texttt{ant}, \texttt{true}\rangle, \langle \texttt{chem}, 60\rangle \quad (42)$$

Finally, upon reaching the nest (where the `nest` gradient is at 0), ants drop the food (*aggregation pattern*), and they are free to look for food again (or to wiggle):

$$\begin{array}{l}\langle \texttt{ant}, \texttt{true}\rangle, \langle \texttt{grad}, \texttt{nest}, 0, M, A\rangle \mapsto \\ \langle \texttt{ant}, \texttt{false}\rangle, \langle \texttt{grad}, \texttt{nest}, 0, M, A\rangle\end{array} \quad (43)$$

With the scheduling algorithm we discussed in Section VI-A1, it is clear that ants will have a certain degree of randomness; for example, when they reach food, they are not obliged to take it (i.e., execute eco-law 39), but they might instead move further (i.e., execute eco-law 38). If it is desirable to specify priority for certain eco-laws (i.e., certain behaviours) over others, then different rates could be set for the eco-laws.

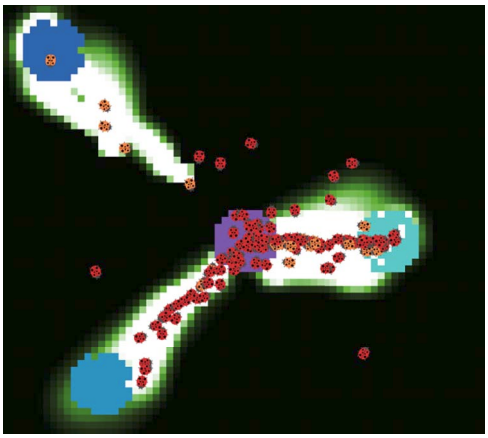

Figure 2. Ant foraging with eco-laws

### B. Evaluation

As for the AODV example, we also implemented both the NetLogo version and eco-law version of the ant foraging model in Repast. Figure 2 shows a screenshot of the Repast version — the top-left, bottom-left and right circles being food sources, the center circle being the nest and white halos indicating the chemical trail. By comparing NetLogo's ant foraging and the eco-law version in terms of convergence, speed of convergence, stability and scalability, we can observe the following:

- *Convergence*—both models converge in the same way. Ants eventually bring all the food back to the nest.
- *Speed of convergence*—this is measured by counting the number of ticks needed until all the food is carried to the nest. Average results for 10 runs are:

| Ants | Ticks: foraging | Ticks: foraging w/eco-laws |
|---|---|---|
| 50 | 1801 +/- 268 | 2342 +/- 233 |
| 100 | 851 +/- 38 | 1114 +/- 79 |
| 150 | 655 +/- 65 | 844 +/- 61 |
| 200 | 632 +/- 62 | 770 +/- 82 |

Ant foraging converges in average faster than ant foraging with eco-laws. This is due to the fact that in ant foraging with eco-laws, all eco-laws are scheduled concurrently with the same priority. No eco-law is supposed a-priori to happen before another one; for example, wiggling and looking for food can happen before checking if there is food to pick up. In the implementation of NetLogo's ant foraging instead, the ant activities have priorities that define a more ordered behaviour. For example, when the ant is looking for food, it checks first at each step if there is food to pick up. If not it can then wiggle looking for food. If there is food, it picks it up and returns to the nest.This can explain in part the increased delays observed in the eco-law based simulation. In addition, ant foraging with

eco-laws uses wiggling not only when ants are looking for food or are returning to the nest but also when they pick up food or are in the nest.

- *Stability*—both models are stable: ants concentrate on following the chemical trails when looking for food and returning to nest when carrying food.
- *Scalability*—the number of ants was varied between 1 and 200. Both models converge in all these cases. The speed of convergence in both cases has values in the same order of magnitude and is proportional to the number of ants.

## VIII. Conclusions

In this article we showed that the SAPERE model, a chemically-inspired model for self-* pervasive systems, is expressive enough to model some existing self-* algorithms. We used SAPERE to develop three different systems. The behaviour has been coded by a set of eco-laws, resulting in simple, clean, compact models with a clear separation of concerns. Simulations showed that the eco-law models behave in a comparable way to the original algorithms with respect to convergence, speed, stability and scalability. However, trying to reproduce *exactly* the algorithms using eco-laws probably introduced some less than efficient modeling patterns. The crowd steering example showed us that when modeling with eco-laws from scratch, the resulting model is both cleaner and more efficient.

We also discussed how in SAPERE models are built as a modular composition of simple patterns, which facilitates scalability, maintainability and reuse. The three different examples were all made by combination of four simple patterns. The pattern classification and modeling activity is still in progress; as new patterns are studied we will establish a more complete library of abstract core behaviours expressed with eco-laws. Also, the language and methodology associated with SAPERE shall evolve to become a richer, easier to use base for systematic development of self-* systems. In particular, the cognitive burden for the adoption of this modeling paradigm should be evaluated.

## Acknowledgement

This work has been supported by the EU-FP7-FET Proactive project SAPERE — Self-aware Pervasive Service Ecosystems, under contract no.256873.

## References

[1] M. Alpuente, M. Falaschi, M. Ramis, and G. Vidal, "A compositional semantics for conditional term rewriting systems," in *Int. Conf. on Computer Languages*. IEEE-CS, 1994.

[2] M. Viroli and F. Zambonelli, "A biochemical approach to adaptive service ecosystems," *Information Sciences*, vol. 180, no. 10, pp. 1876–1892, 2010.

[3] SAPERE project, "The SAPERE approach," http://www.sapere-project.eu/sapere-approach.

[4] S. Montagna, M. Viroli, M. Risoldi, D. Pianini, and G. Di Marzo Serugendo, "Self-organising pervasive ecosystems: A crowd evacuation example," in *3rd International Workshop on Software Engineering for Resilient Systems*. Springer, 2010, in press.

[5] J. L. Fernandez-Marquez, J. L. Arcos, G. Di Marzo Serugendo, M. Viroli, and S. Montagna, "Description and composition of bio-inspired design patterns: the gradient case," in *3rd Workshop on Bio-Inspired and Self-* Algorithms for Distributed Systems*. ACM, 2011, pp. 25–32.

[6] J. L. Fernandez-Marquez, J. L. Arcos, G. Di Marzo Serugendo, and M. Casadei, "Description and composition of bio-inspired design patterns: the gossip case," in *8th International Conference and Workshop on Engineering of Autonomic and Autonomous Systems*. IEEE-CS, 2011, pp. 87–96.

[7] I. Schaefer and A. Poetzsch-Heffter, "Using abstraction in modular verification of synchronous adaptive systems," in *Trustworthy Software*, ser. OASICS, S. Autexier, S. Merz, L. W. N. van der Torre, R. Wilhelm, and P. Wolper, Eds., vol. 3. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[8] G. Di Marzo Serugendo, "Robustness and dependability of self-organizing systems - a safety engineering perspective," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science, R. Guerraoui and F. Petit, Eds. Springer, 2009, vol. 5873, pp. 254–268.

[9] M. North, T. Howe, N. Collier, and J. Vos, "A declarative model assembly infrastructure for verification and validation," in *Advancing Social Simulation: The First World Congress*, S. Takahashi, D. Sallach, and J. Rouchier, Eds. Springer, 2007.

[10] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," in *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, ser. WMCSA '99. Washington, DC, USA: IEEE-CS, 1999.

[11] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc on-demand distance vector (AODV) routing," 2003, Network Working group RFC #3561, available at http://www.ietf.org/rfc/rfc3561.txt.

[12] U. Wilensky, "NetLogo ants model," 1997, Center for Connected Learning and Computer-Based Modeling. Northwestern University, Evanston. http://ccl.northwestern.edu/netlogo/models/Ants.