

Describing Self-Organizing Software with Design Patterns: A Reverse Engineering Experience

Paul L. Snyder, Giuseppe Valetto
Drexel University
Department of Computer Science
Philadelphia, Pennsylvania, USA

Jose Luis Fernandez-Marquez,
Giovanna Di Marzo Serugendo
University of Geneva, CUI
Carouge, SWITZERLAND

Abstract—Investigations of self-organizing mechanisms, often inspired by phenomena in natural or societal systems, have yielded a wealth of techniques for the self-adaptation of complex, large- and ultra-large-scale software systems.

The principled design of self-adaptive software using principles of self-organization remains challenging. Several studies have approached this problem by proposing design patterns for self-organization. In this paper, we present the results of applying a catalog of biologically inspired design patterns to Mycoload, a self-organizing system for clustering and load balancing in decentralized service networks.

We reverse-engineered Mycoload, obtaining a design that isolates instances of several patterns. This exercise allowed us to identify additional reusable self-organization mechanisms, which we have also abstracted out as design patterns: SPECIALIZATION, which we present here for the first time, and a generalized form of COLLECTIVE SORT. The pattern-based design also led to a better understanding of the relationships among the multiple self-organizing mechanisms that together determine the emergent dynamics of Mycoload.

Keywords—self-organization; design patterns; bio-inspired algorithms; design modeling.

I. INTRODUCTION

Modern computing and communications systems continue to expand in scale. We witness more and more examples of ubiquitous computing systems, social networks, wireless sensor networks, peer-to-peer overlays, and many others, which encompass huge numbers of components on heterogeneous devices, often under multiple ownerships.

These systems pose significant challenges. In order to ensure critical runtime properties such as scalability, fault tolerance, performance, responsiveness, self-adaptive management is highly desirable. However, centralized “command-and-control” approaches to autonomous management quickly become impractical because of issues of scale, complexity, extreme dynamism, and indistinct boundaries. Thus, researchers have been taken principles of self-organization that have been studied in the field of Complex Adaptive Systems (CAS) [1], and applied them to the decentralized self-adaptation of software systems. Many self-organizing systems have appealing characteristics: robustness, resilience, the ability to adapt to environmental

changes, and the ability to achieve complex behavior using a simple set of local rules [2].

While techniques of self-organization have proven effective in enabling system-wide adaptations of large-scale software in a number of domains, it remains challenging to represent and to reason about such these mechanisms [3]. We need to develop modular, reusable design primitives in order to achieve principled and systematic engineering of self-organized adaptation in software systems.

To address this problem, several attempts have been made to identify and collect design patterns for self-organization [4], [5]. Previous work by Fernandez-Marquez *et al.* [6] has proposed a catalog of bio-inspired design patterns, defining a hierarchy of patterns where basic patterns are used as elements of composed, higher-level dynamics. The motivation of that work is to organize the growing body of knowledge in the area, and foster modular, reusable design of self-organizing software.

In this paper, we explore the practical implications and benefit of those patterns. To that end, we have decomposed and modeled the design of a decentralized service network, called Mycoload [7], running on top of a biologically inspired peer-to-peer network developed by Snyder *et al.* [8]. This reverse engineering exercise has yielded multiple contributions: (1) it has made evident how a number of mechanisms that we have implemented in Mycoload are instances of the patterns in our catalog, demonstrating how design pattern abstractions can provide a greater understanding of a real-world, complex software system that uses self-organization principles for its self-adaptation; (2) it has shown how pattern-based decomposition can expose otherwise implicit interactions among those patterns, interactions that play an important role in determining the overall dynamics of a system; and (3) it has allowed the isolation of other reusable self-organizing mechanisms that we can now introduce in our catalog as design patterns.

Before plunging into a detailed discussion of our reverse-engineering case study, and the new and existing patterns that play a role in it (see Sections V and VI), we review the state of the art in this area, in Section II, and offer an overview of our own background work on a self-organized

design pattern catalog (Section III) and the Mycoload system with its self-organized dynamics (Section IV).

II. RELATED WORK

Researchers have repeatedly taken inspiration from biological and other natural self-organizing systems, and have identified adaptive mechanisms that can be mimicked in computing systems. This approach allows results that often go beyond the possibilities of centralized control in many scenarios [9], [10]. However, these self-organizing mechanisms are typically approached in an ad-hoc or highly application-specific manner, which prevents their systematic reuse, and hampers their application to recurrent problems across different domains.

Among the works that attempt to define design patterns for self-organizing software, some focus on the discovery and definition of a single pattern [11], [12]; others propose concrete implementation descriptions [13]; yet others catalog multiple patterns [14], [4], [5], [15]. Our previous work [6], discussed in Section III, is most similar to the latter. One feature that sets our catalog of bio-inspired patterns apart from other efforts is the organization of the patterns into layers, and the documentation of composition relationships between patterns in those layers. As pointed out by Parunak and Brueckner [3], the definition of composition and decomposition relationships is important, because—while it seems clear that certain self-organization mechanisms can be obtained from finer-grained ones — which ones should be used as primitives, and how they should be combined is often not clear, and almost never explicitly codified. Some interesting work on composition has been performed by Sudeikat and Renz [14], who take the approach of looking for congruent feedback loops among multiple self-organizing mechanisms.

In general, there is still a lack of guidance on how to use identified self-organizing design patterns in the engineering of larger self-adaptive systems that rely on multiple mechanisms. The work we present may be the first that tries to apply patterns exhaustively to model an existing complex self-organized system, in order to shed a better light on its self-adaptive dynamics. In that sense, it is similar to the work by Ramirez *et al.* [16], which analyzed a list of adaptation design patterns, and re-engineered an application using those patterns. The authors could identify several advantages of using such design patterns. Since that work focused on adaptation design patterns, we consider that a similar effort must be done also for specifically evaluating self-organizing design patterns, and fostering their reuse.

III. SELF-ORGANIZATION DESIGN PATTERNS

In previous work, Fernandez-Marquez *et al.* [6], presented a catalog of self-organization design patterns (largely inspired from mechanisms observed in biology), and analyzed the relations between them. One contribution is a three-layer

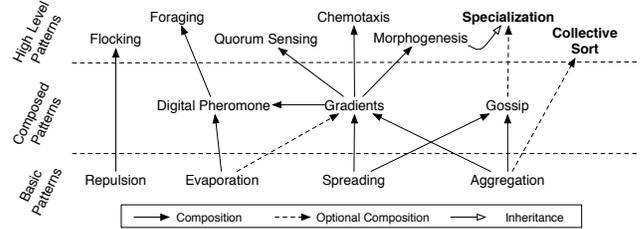


Figure 1: Self-Organization Design Patterns

classification (basic, composed and high-level patterns), shown in Figure 1. This classification scheme also includes relations among different self-organizing mechanisms, such as composition and usage: patterns in the lower layers provide building blocks for more sophisticated patterns at the higher layers. A good example is the DIGITAL PHEROMONE pattern, which uses a SPREADING mechanism to disperse the pheromones over the environment, an AGGREGATION mechanism to combine multiple pheromone concentrations at a location, and an EVAPORATION mechanism to cause pheromone concentrations to decay over time.

The catalog shows how a limited number of basic mechanisms are at the basis of a large set of powerful self-organizing dynamics that have been examined in the literature (including gossip, gradients, and morphogenesis), which can also be expressed as design patterns. Our classification scheme thus offers a key to understand complex as well as basic self-organization mechanisms. Most importantly, it enables the design of self-organizing applications in terms of well-defined, modular and reusable blocks.

We now briefly review a selected set of these patterns, the ones that will be used in the reverse engineering of the Mycoload system (Section VI). Due to space limitations, full descriptions are not presented here, but can be found in [6].

1) **SPREADING:** The SPREADING pattern is a basic pattern for information dissemination or diffusion, aiming at incrementing the global knowledge of the system using only local interactions.

In systems where agents act and perceive in limited neighborhoods, their reasoning may suffer from a lack of knowledge about the global system. To alleviate this, a copy of information received or held by an agent is sent to neighbors and propagated over the network. Information spreads progressively through the system and reduces agents' lack of knowledge while keeping the constraint of local interaction.

2) **AGGREGATION:** AGGREGATION is a basic pattern for information fusion, aiming to reduce excess information, while retaining enough meaningful information to enable local decision-making.

In large systems, excesses of information produced by the agents may produce network and memory overloads, and local actions must be performed in order to reduce the total amount of information and render it into a meaningful form. Aggregation consists in locally applying a fusion operator

to process and synthesize information. Fusion operators are classified into four different groups: filters, transformers, mergers and aggregators [17].

3) **EVAPORATION**: EVAPORATION is a basic pattern that helps to deal with dynamic environments where information used by agents can become outdated, or where the freshness of information is crucial for the agents' decisions.

In cases where outdated information cannot be directly detected and it needs to be removed, or where its detection involves a cost that needs to be avoided, evaporation progressively reduces its relevance. Thus, recent information can be easily determined to be more relevant than information processed some time ago, and agent choices can then be informed by the freshness of the information, enabling correct responses to dynamic environments.

4) **GOSSIP**: Its goal is to obtain an agreement about the value of some parameters in the system in a decentralized way. Agents in the system collaborate to progressively reach this agreement. Each contributes by aggregating its own knowledge with its neighbors' knowledge and by spreading this aggregated knowledge. Thus, GOSSIP is composed of the SPREADING and AGGREGATION patterns.

5) **GRADIENT**: The GRADIENT pattern is an extension of the SPREADING pattern. The information is propagated in such a way that helps determining the sender's distance. Either a distance attribute is added to the information, or the value of the information is modified such that it reflects its concentration—higher values indicating the sender is closer, as in ant-inspired pheromone models. Additionally, the GRADIENT pattern uses AGGREGATION to merge gradients created by different agents or to merge gradients coming from the same agent through different paths.

As a gradient is created, information spreads from the location where it is initially created or deposited and aggregates when it meets other information. During spreading, additional information about the sender's distance and direction is provided, either through a distance value (incremented or decremented) or by modifying the information to represent its concentration (as when information that is further away is modeled using lower concentrations relative to closely-situated information).

IV. MYCOLOAD

Mycoload [7] is a self-organizing system for clustering and load-balancing in unstructured peer-to-peer networks. Mycoload is an extension to Myconet [8], a protocol for constructing superpeer-based overlays, inspired by the growth patterns of fungal root systems, called *hyphae*.

In this section, we provide some background on the characteristics and behavior of Mycoload in order to introduce and motivate the SPECIALIZATION and COLLECTIVE SORT patterns presented in Section V, and to inform the in-depth discussion of its design, expressed via self-organization patterns, which takes place in Section VI. A detailed description

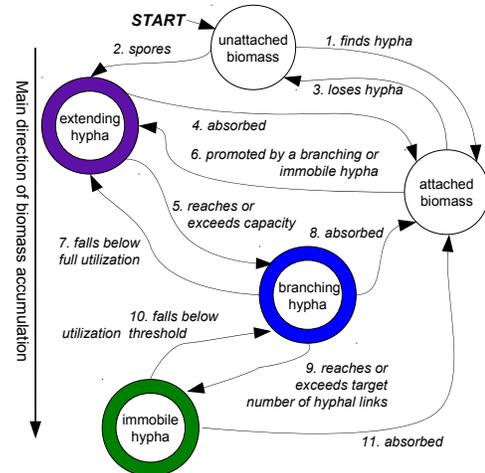


Figure 2: Mycoload protocol states and transitions.

of protocol rules or results for Myconet and Mycoload are out of the scope of this paper, but interested readers can find those details in [8] and [7].

Mycoload is built as an application on top of the Myconet overlay protocol. Myconet uses a fungal metaphor in which superpeers are considered to be hyphae, and ordinary, non-superpeer nodes are considered to be *biomass* (that is, nutrients) to be moved among the hyphae as needed. Mycoload peers use an abstract capacity measure to capture heterogenous properties, with higher capacities representing better suitability to act as a superpeer and provide services to other peers.

Myconet constructs and maintains a strongly interconnected overlay that quickly converges to an optimal number of superpeers. It rapidly self-heals, repairing damage caused by failed peers, and dynamically adjusts the network topology to changing conditions. One of its distinguishing characteristics is that superpeers move through a hierarchical set of protocol states. Each state has different roles and responsibilities in maintaining the overlay, and peers are promoted or demoted to those states based on how well they are able to carry out those responsibilities. The states and transitions are shown in the diagram in Figure 2. When promoted from biomass to the first, *extending* hyphal state, superpeers are mainly focused on exploration, acting as connection points for new biomass peers entering the network. Peers will remain in this state until connected to at least one other peer providing the same service type. *Branching* hyphae are superpeers that have reached their target level of utilization (*i.e.*, they have connected to sufficient biomass peers to reach a level of utilization proportional to their capacity attribute); they manage the number of extending superpeers in the network while growing new hyphal interconnections. Finally, *immobile* hyphae have reached levels of full biomass connection and a target number of hyphal interconnections, and are considered to be relatively stable, having remained in the network long enough to transition through all the

previous protocol states.

Mycoload extends the Myconet protocol with rules that aim at constructing clusters of peers offering the same service type. Self-organized load balancing is then applied to the peers within each cluster (following a variant of the algorithm in [18]), to efficiently distribute requests for services of each type among the peers' job queues. Load balancing is performed between same service-type neighbors within a cluster, with local exchanges striving to keep queue lengths proportional to the capacity of each peer. When a load-balancing operation is performed, a peer selects a random same-type superpeer neighbor; as biomass peers have only a single neighbor, they will be the ones to initiate the load-balancing operations with their parent. This way, load-balancing operations tend to occur preferentially between the more-capable superpeers, which helps ensure that jobs will be balanced throughout the cluster. "Ideal" queue lengths are calculated by determining the total number of jobs in both queues and dividing the jobs proportionally based on the peers' capacities.

In its current implementation, each Mycoload peer offers only a single service type, and each cluster in the overlay collects peers of that type. The process of designing an extension to Mycoload where peers can host multiple types of services is the original motivation for modeling the current application as a set of patterns that tease out the different elements of its self-organizing dynamics. That pattern-based analysis is presented in Section VI, and yielded the identification of two additional design pattern for self-organization, which we discuss in the next section.

V. PATTERN DISCOVERY IN MYCOLOAD

When we analyzed Mycoload, we identified several instances of patterns that are included in [6]. We also isolated two important aspects of its self-organizing behavior that are not encompassed by other patterns but can themselves become reusable mechanisms. We have been able to abstract those mechanisms as instances of two design patterns: the first one, SPECIALIZATION has not been presented as a self-organization pattern before in the literature, as far as the authors know; the second one is a generalization of the COLLECTIVE SORT in pattern [4].

In Section VI, we will discuss the role of SPECIALIZATION and COLLECTIVE SORT in the overall Mycoload design; we will also highlight their relationships and interactions with the other patterns from the catalog. In the remainder of this section, instead, we specify these pattern individually, following the self-organization design pattern structure used in [6].

A. Specialization Pattern

Specialization is a mechanism widely used in complex systems for achieving improved efficiency by exploiting the natural heterogeneity of the entities taking part in the

system. Through SPECIALIZATION, each individual entity is assigned a specific role depending on its capabilities and contextual local information. This assignment can be considered as "closed" specialization when the set of possible roles is known a priori, or "open" specialization otherwise. A useful survey of specialization in self-organizing systems can be found in [19].

According to the SPECIALIZATION patterns, system entities will change the rules under which they operate, depending on features or properties of the entity itself, or contextual information from its environment and neighbors. For example, a computer with high amounts of memory available could store information on behalf of nodes with low memory; a computer with sensors could provide sensed information to other computers; a network node with plenty of bandwidth connections could be elected to act as a router for traffic transmitted by other nodes, and so on. The assumption of specialized roles may be influenced, or may need to be further modulated, to adapt to changing system conditions and requirements in the system.

Name: SPECIALIZATION

Aliases: None to our knowledge.

Problem: Global optimization of system efficiency by increasing or decreasing the contributions of individual entities or by otherwise changing the rules under which those entities operate.

Solution: Individual entities are assigned a specific role or set of behavioral rules depending on their capabilities and contextual local information. SPECIALIZATION optimizes entities' contributions in order to increase the overall performance of the system.

Inspiration: The specialization process appears in many macro- and micro-level systems. Some examples are the specialization of cells in a human body or the specialization of individual humans to fill particular roles in society.

Forces: Depending on how the contextual information used for making decisions regarding specialization is acquired and which patterns are used in transferring and maintaining this information, different trade-offs can appear. The most common patterns are SPREADING and AGGREGATION (see the forces discussed in their pattern descriptions [6] for more details.) In general, though, the information used by SPECIALIZATION can come from any other self-organizing mechanisms or a combination of those mechanisms, and their dynamics will influence and possibly be mutually influence the resulting specializations.

Entities: The entities participating in the SPECIALIZATION pattern are: (1) software agents that modify their behavior depending on their capabilities (or the capabilities of their hosts) and environmental information (e.g. external requirements); (2) hosts that provide sensors, memory, communication capability, computational power, etc. to the software agents; and finally (3) Environment, all that is external to the hosts (e.g. the space where host are located, external requirements that are injected in the system, etc...)

Dynamics: Agents retrieve contextual information from their own knowledge and from their neighbor agents, or from the environment by using sensors or querying an externally implemented environmental model. To describe the dynamics we use the same notation as in [6], where information contained in the system is modelled as a tuple $\langle L, C \rangle$, where L is the location where the information is stored (possibly within an agent or maintained by

an external middleware), and C is its current content—e.g., in the form of a list with one or more arguments of different types, such as numbers, strings or structured data, according to the application-specific information content. Transition rules resemble chemical reactions between patterns of tuples, where (i) the left-hand side (reagents) specifies which tuples are involved in the transition rule: they will be removed as an effect of the rule execution; (ii) the right-hand side (products) specifies which tuples are accordingly to be inserted back in the specified locations: they might be new tuples, transformation of one or more reagents or even unchanged reagents; and (iii) rate r is a rate, indicating the speed/frequency at which the rule is to be fired (that is, its scheduling policy).

$$\text{state_evolution} :: \langle L, [cInf, State, C] \rangle \xrightarrow{r_{sp}} \langle L, [cInf, State', C] \rangle$$

where $State' = \pi(cInf, State, C)$

In the above rule, $cInf$ is the contextual information accessible to the agent, $State$ is its previous role before the specialization occurs (i.e., the set of rules under which it operates), $State'$ is the new role (and consequent set of rules) adopted as a result of the specialization process, and π is a function that produces this new state from the given contextual information, current agent state, and any local information.

Environment: The hosts must have different features or the system must display other heterogeneities (for example, in the distribution of agents in different locations) that allow SPECIALIZATION to assign an appropriate role based on those features and the contextual information.

Implementation: We have identified two different implementations: (1) An agent decides to change its role in the system by taking into account the capabilities of the local environment where it resides and acquired contextual information (e.g., the system's requirements); and (2) An agent is positioned to determine that one of its neighbors should adopt a new role. An example of the second case is when one node is providing services to other nodes in the system but it is reaching the maximum number of clients; in such a case the node can replicate the information served to a new node (selected according to some suitability measure from among other nodes in its neighborhood) and target it to assume the role of an additional service provider.

Two types of rules can be used to drive the specialization of agents, determining which role an individual adopts depending on its capabilities and context: fixed rules, and adaptive rules. Fixed rules are defined by developers at design time, agents switch among behaviors from a static set. Adaptive rules may be changed by the agents during run time in order to contribute to the optimization of the global system behaviour. Evolutionary approaches have been used in the field of autonomic computing to establish sets of norms, policies or rules that drive the system to the desired emergent behaviour, even when environmental changes occur. A possible implementation was introduced in [20], which uses a distributed genetic algorithm. In that approach, each agent participating in the system performs local evaluations and adjustments that are then shared with other agents using spreading mechanisms.

Known Uses: Specialization has been used by a large number of self-organizing applications. Examples include: (1) Overlay networks where some nodes decide to become routers based on their available resources and their connectivity with other nearby nodes [21] (2) Aiming to localize diffuse event sources in dynamic environments using large scale wireless sensor networks, agents change their roles in order to locate and track diffuse event sources [22]. (3) To balance the load among nodes with different services, Mycoload [7], builds a superpeer topology where more powerful nodes adopt several different, specialized roles in the

creation and maintainance of the overlay. (4) [23] describes a robust process for shape formation on a sheet of identically programmed agents (origami) where the heterogeneity of the agents comes from their location.

Consequences: Specialization locally increases or decreases the contribution of individual nodes, improving the global efficiency of the system.

Related Patterns: In the MORPHOGENESIS pattern, the role of the agents changes depending on their relative positions, typically communicated via a GRADIENT. Thus, MORPHOGENESIS is of the same family but more specific than SPECIALIZATION.

A more detailed discussion of the role of SPECIALIZATION within the Mycoload application is provided in Section VI, below.

B. Generalized Collective Sort Pattern

In this section we discuss the COLLECTIVE SORT self-organization design pattern, which is used to gather similar elements or data in a system into similar vicinities.

Collective sort is a clustering mechanism that enables segregation or relocation of entities into similar-type groupings within the context of a collection of system elements according to some property of the entities or requirement of the system [24], [25], [26]. Self-organizing algorithms for collective sorting have been developed based on observations of biological phenomena, particularly the processes of brood sorting and cemetery formation by social insects [27], [28].

The mechanism discussed in this pattern is a generalization of the biologically inspired collective sorting proposed as a design pattern for tuple spaces by Gardelli *et al.* [4], [29] and analyzed as an environmental coordination mechanism by Sudeikat and Renz [26]. The usual formulation of the collective sort algorithm assumes a case where active agents relocate inactive data items; the COLLECTIVE SORT pattern presented in this paper extends this to include cases where the agents themselves may be the entities to be grouped, or where different environmental abstractions are being used. These variants are discussed in the Implementation section of the pattern description, below.

Name: COLLECTIVE SORT

Aliases: Brood Sorting, Cemetery Formation, Collective Clustering

Problem: A system contains a number of scattered data or entities that need to be brought into relative proximity with other similar data or entities.

Solution: Individual agents move through an environment, encountering data items as they travel. By picking up and dropping these items based on local heuristics, elements with similar properties are progressively gathered into homogeneous groups or clusters.

Inspiration: Brood sorting and cemetery formation by social insects [27], [28]

Forces: This pattern starts from a disordered arrangement of entities within an environment and progressively reduces that disorder. Thus, it is affected by entity distribution in the environment, and other other forces that act to change the location of these entities. The function used by entities to evaluate local density of entities (as well as an entity's range of perception) will affect the outcome of the sort. In particular, a small range may induce

the formation of multiple small collections. The choice of rules and probabilities for the picking up and dropping of entities may also influence the speed of convergence or the resulting topological distribution.

Entities: The entities associated with the COLLECTIVE SORT pattern are: (1) data items that have an associated property for which similarity can be assessed; and (2) active agents that are able to examine and relocate data items of type (1). Note that, depending on the implementation, (1) and (2) may be the same entities. For example, the active agents may themselves possess the property that is the subject of sorting, and hence the emergent order will be expressed by the arrangement of the agents themselves.

Dynamics: This pattern relies on three rules which, together, tend to progressively relocate similar elements into similar vicinities: a *Movement Rule* that relocates agents within a set of candidate locations, a *Pick-Up Rule* that connects an agent to an element so it can be moved, and a *Drop Rule* that leaves a held element at a current location. These rules are followed by the active agents. The Movement rule is frequently implemented as random exploration, but could also take advantage of available contextual information if appropriate. The Pick-Up and Drop rules select entities to be clustered when they are encountered, tending to remove elements from areas of high diversity and deposit them in areas of low diversity. Two approaches to applying these general rules are discussed in the Implementation section, below.

Environment: The environment provides the context within which the proximity of data items is interpreted. Thus, entities must have a concept of location, be able to change location, and be able to detect other nearby entities within that environment; they must also have a means of evaluating the similarity of data items thus detected.

Implementation: Different implementations are possible. We here discuss two scenarios: a distributed tuple space and a peer-to-peer network.

In a tuple space implementation active agents typically move from tuple space to tuple space, carrying tuples with them. An example of rules implementing a biologically inspired variant of COLLECTIVE SORT in this case (derived from [30]) is:

1. *Movement Rule:* Agents explore the environment, encountering data items as they move. The choice of movement strategy is frequently random, but may also be informed by other information, such as using a CHEMOTAXIS strategy. An example of a random exploration rule for an agent is:

$$\text{random_movement} :: \langle L, C \rangle \xrightarrow{r_{\text{move}}} \langle L', C \rangle$$

where $L' = \text{random}(\text{NEIGHBORHOOD}(L))$

2. *Pick-Up Rule:* Agents “transport” data items from place to place, tending to move them from areas with lesser concentration to areas with greater concentration of items of the correct type. This rule may rely on direct observation of data items in a vicinity (as with observation of neighbors in FLOCKING); it may rely on AGGREGATION to estimate the local density of data items of a particular type; or the agent may make its own estimate by maintaining a memory of recently encountered data items. A general way to express how pick-up occurs is through a probabilistic function of the density of data items. For example, the probability of picking up an encountered data element of type t at location L might be:

$$P_{\text{pick_up}}^t = \left(\frac{k_{\text{pick_up}}}{k_{\text{pick_up}} + d^t(L)} \right)^2$$

where d^t is the estimated density of data items of type t at location L , and $k_{\text{pick_up}}$ is a constant; the associated rule when the agent encounters element e^t of type t in the local environment E would

be executed with probability $P_{\text{pick_up}}$:

$$\text{pick_up} :: \langle \langle L, C \rangle, E \rangle \xrightarrow{r_{\text{pick_up}}} \langle \langle L, C \cup \{e^t\} \rangle, E - \{e^t\} \rangle$$

3. *Drop Rule:* Similar to the Pick-Up Rule, agents will drop items when they estimate that the local vicinity is propitious. The probability of dropping a held e^t element of type t at location L with environment E might be:

$$P_{\text{drop}}^t = \left(\frac{d^t(L)}{k_d + d^t(L)} \right)^2$$

and the associated drop rule, executed by the agent with P_{drop} , would be:

$$\text{drop} :: \langle \langle L, C \rangle, E \rangle \xrightarrow{r_{\text{drop}}} \langle \langle L, C - e^t \rangle, E \cup \{e^t\} \rangle$$

In a more general view of COLLECTIVE SORT, an abstract notion of grouping can be used to apply the same strategy in a scenario where the environment is defined by a pattern of neighbor relationships between nodes (composing a graph, as in a P2P network), and where the nodes themselves are labelled with some property (e.g., a node type) upon which clustering should be performed. In this dynamic graph scenario, “movement” is considered to be selecting a candidate location for growing a new neighbor relationship, “picking up” is adding a new neighbor, and “dropping” is severing an existing neighbor relationship.

1. *Movement Rule:* Agents explore randomly by selecting a potential new neighbor from a set of possible candidates. In many P2P networks, and in Mycoload, this set is maintained by a separate mechanism that implements the GOSSIP pattern, and thus provides each node with fresh samples of candidate non-neighbor nodes. For a node V with neighbor set $N(V)$, a new possible neighbor $Cand$ is selected:

$$\text{graph_movement} :: \langle V, N(V) \rangle \xrightarrow{r_{\text{move}}} \langle V, N(V), Cand \rangle$$

where $Cand = \text{random}(\text{CANDIDATES}(V))$ and $Cand \notin N(V)$

Note that this may also result in V finding a cluster of its own type (if the new neighbor W is of the same type) if it was not already in one, or finding a connecting path for two disconnected same-type sub-clusters.

2. *Pick-Up Rule:* Once the movement rule has selected a new possible neighbor $Cand$, the agent may add it as a new neighbor to itself

$$\text{graph_pick_up} :: \langle V, N(V), Cand \rangle \xrightarrow{r_{\text{pick_up}}} \langle V, N(V) \cup \{Cand\} \rangle$$

In Mycoload, for example, the pick-up rule will be executed if (a) V does not currently have a neighbor that is its same type, (b) if V has under a certain number of different-type neighbors, or (c) execute anyway with a small probability to prevent the system from settling into a local optimum. Thus, agents will wander until they find a cluster of the same type (whether by encountering it by moving or dropping in place by another agent), but will also try to keep connections to neighbors of other types in order to help other nodes move toward an appropriate cluster. The pick-up rate thus declines as the nodes converge toward clusters.

3. *Drop Rule:* Dropping for a node V is performed by randomly selecting a neighbor $W \in N(V)$ (where the type of W with neighbor set $N(W)$ is different from the type of V) and, if V also has a neighbor $U \in N(V)$ with the same type as W , by transferring W to become a neighbor of U :

$$\text{graph_drop} :: \langle \langle V, N(V) \rangle, \langle W, N(W) \rangle, \langle U, N(U) \rangle \rangle \xrightarrow{r_{\text{drop}}} \langle \langle V, N(V) - W \rangle, \langle W, N(W) \cup \{U\} \rangle, \langle U, N(U) \cup \{W\} \rangle \rangle$$

A more detailed discussion of the role of COLLECTIVE SORT

within the Mycoload application is provided in Section VI, below.

Known Uses: Collective sort and brood sorting-inspired approaches to distributed self-organizing systems have been applied to several problems areas: (1) Collecting similar tuples from a distributed tuple system into a single tuple space [29]. Of interest is Casadei *et al.*'s approach using "noise" tuples to implement a simulated annealing-type approach to avoiding local optima [31]. (2) Collective sorting as a coordination mechanism for swarms of self-organized robots, proposed as early as 1991 by Deneubourg *et al.* [32] and extended by later researchers [25]. (3) Storage and retrieval of Semantic Web documents. Presenting such a scenario, Muhleisen *et al.* [24] discuss in particular the role of similarity metric selection in collective sorting. (4) Intrusion detection. Sudeikat and Renz [26] identify brood sorting as suitable for providing a portion of the self-organizing dynamics for a stigmergic IDS. (5) Clustering of same-type nodes in peer-to-peer networks. Mycoload [7] uses a collective sort approach to build clusters of peers offering the same service types; the specific role of the collective sort mechanisms is discussed in this paper.

Consequences: Collective sort enables clustering of data or other entities into groups of similar type. The resulting order may increase efficiency of operations on this data.

Related Patterns: AGGREGATION is often used to estimate local density of data items. SPREADING and GRADIENT may be used to disseminate information about the density of particular kinds of data, and CHEMOTAXIS may be used to guide agent movement. SPECIALIZATION may be used to select specific items among the ones in the resulting groupings for particular roles.

VI. REVERSE ENGINEERING OF MYCOLOAD WITH PATTERNS

In this section, we analyze an existing self-organizing system, Mycoload. The overall behavior of Mycoload emerges as the result of the combined action of multiple self-organizing mechanisms. These were implemented on the basis of a biological metaphor applied to peer-to-peer overlays, and underwent further incremental refinement and tuning over time, to accommodate the requirement of Mycoload as a *decentralized service network*, as discussed in [7].

We set out to describe the complex and composite self-adaptive behavior of Mycoload, approaching this task as a reverse engineering project with the goal of identifying, isolating, and modularizing its various self-organizing mechanisms. The outcome is a view of its design modeled in terms of the design patterns discussed in Sections III and V.

The primary goal of Mycoload is to *maximize throughput by balancing load queues for multiple services across many peers in a decentralized service network with heterogeneous capabilities*. We decomposed this goal into a set of primary features provided by the system. First, in order to construct an efficient overlay network, (A) *peers discover appropriate candidates to become neighbors when entering the system and while maintaining the overlay*; and (B) *the overlay exploits the heterogeneity of the system by promoting higher-capacity peers to specialized roles with higher responsibilities in the maintenance of the overlay*. The overlay also (C) *collects peers offering the same service into connected*

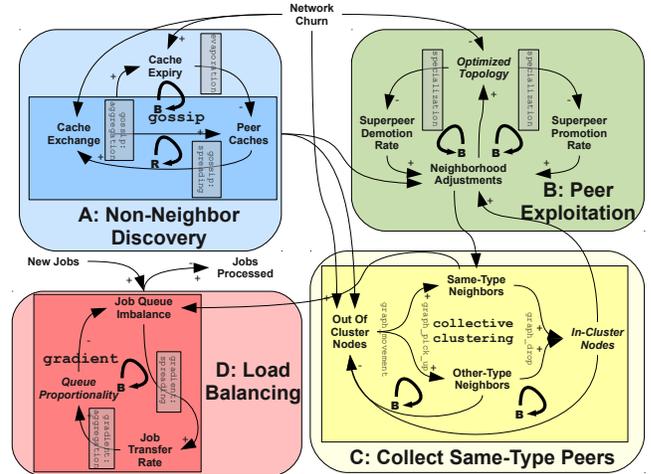


Figure 3: Annotated CLD showing interrelations and patterns of self-organizing mechanisms in Mycoload

subgroups, so that they can interchange jobs with other peers in the same subgroup. Finally, Mycoload (D) *shifts jobs between peers such that peers with higher capacity are given proportionally more work*, and thus performs load balancing of job requests.

The breakdown of Mycoload according to the four major features above is shown in Figure 3, which depicts representation of the Mycoload design as an annotated causal loop diagram (CLD). CLDs are often used to describe and reason about Complex Adaptive Systems, and have been used by Sudeikat and Renz [14] for analyzing self-organizing coordination mechanisms. We have adopted a CLD view of our design because, in comparison with other design modeling facilities—such as UML structural diagrams (*e.g.*, component diagrams) or interaction diagrams (*e.g.*, sequence diagrams)—CLDs make explicit the feedback loops which exist in self-organized systems, and which are essential to their functionality.

Our reverse-engineering exercise has shown how multiple feedback loops are present in Myconet, and how they interact with one another. Some loops correspond one-to-one to self-organization design patterns that we have been able to recognize and isolate; in other cases, patterns—especially those from the basic layer of our catalog—are best associated with transitions (arrows) within a loop; in yet other cases, certain transitions can be associated to one of the transition rules that we have defined in the body of a pattern specification (as in block (C) of the figure). We highlighted with different textual or graphic annotations each of those cases in our diagram: the Myconet features discussed below are highlighted by colored, rounded rectangles around the corresponding portions of the diagram. High-level patterns are shown as squared rectangles labeled with a monospace font. Grey rectangles oriented vertically and associated with transitions indicate basic patterns, and larger shaded rectangles cover a range of states associated with a higher-level

pattern. Vertically oriented monospace text without a grey box indicates a rule that is part of a pattern.

The "by-feature" decomposition of the design proposed in Figure 3 has allowed us to see where each loop (and therefore each design pattern) operates in the system, and how it influences other loops and other features. In the remainder of this Section, we discuss each area of this composed CLD diagram.

(A) *Discovery of candidate neighbor peers*

Peers need to be able to discover other peers when entering the system in order to create new, appropriate neighbor relationships for building and maintaining the overlay. Ideally, a system with perfect knowledge would be able to select a suitable peer from anywhere in the overlay. In practice, given the decentralized and dynamic nature of large-scale peer-to-peer overlays, peers must achieve this through a mechanism that works locally and maintains a limited set of data.

One popular mechanism for that is the GOSSIP pattern, which allows peers to share knowledge about other active peers, outside of the context of established neighbor relationships. With each interaction, pairs of peers contribute to a joint agreement on a set of nodes that are likely to be fresh (*i.e.*, likely live and participating in the overlay); this information is then spread to other nodes through subsequent exchanges. The GOSSIP mechanism is composed of two sub-patterns, SPREADING and AGGREGATION. Both mechanisms, in this case, operate on a list of known peers that is maintained at each local node. Each peer periodically selects a known peer and sends a copy of its entire list (SPREADING). Upon receiving such a message, the recipient combines that list with its own (AGGREGATION), along with an entry for the sending peer timestamped with the current time. In order to prevent the lists from growing indefinitely (and to eventually remove peers that are no longer participating in the gossip exchanges), peers are removed from the list via an EVAPORATION mechanism. Entries with the oldest time are progressively discarded, keeping the list at a fixed length.

In Mycoload, GOSSIP as well as all of its component patterns are implemented by the Newscast protocol [33], which Mycoload adopts as an off-the-shelf component in order to fulfill feature (A).

(B) *Peer Exploitation*

Mycoload is designed to exploit communities of peers with highly heterogeneous capabilities and resources (that is, where some peers are much more powerful and have a much higher capacity than others). Heterogeneity brings about an opportunity for improved performance, with a limited number of peers with more capabilities tasked to perform certain advanced functions; this is a basic functioning principle for

all super-peer overlays, and represents a typical use case for the SPECIALIZATION pattern.

In Mycoload, each peer considers its own status and the situation in its neighborhood (primarily, its number of neighbors relative to an ideal capacity, a measure based on its available resources, and its own capacity compared to that of its neighbors). Each peer uses this local information to determine whether to adopt a new role (as a superpeer in one of the hyphal stages described in section IV), and switching the set of protocol rules it uses to manage neighbor relationships accordingly.

The information considered by a peer when making the decision on whether to adopt a new, specialized state may also include messages from other peers (in this case, superpeers with a broader view of the network); these are received "hints" that the peer is well-positioned to be promoted and serve as a new superpeer, or that it has now become redundant, and it can reduce inefficiency by demoting itself. A full description topology maintenance rules followed in each specialized state cannot appear here due to space limitations, but can be found in [7].

(C) *Collect peers offering similar services into groups*

In the service network scenario targeted by Mycoload, peers may offer different services, and jobs of a certain type may only be processed by peers offering a service of the same type. In order to exchange jobs and balance their respective loads, peers of similar type need to be able to find, group with, and collaborate with each other. To enable that, the COLLECTIVE SORT pattern operates to adjust the topology of the overlay in such a way that neighborhoods of peers with the same service type will emerge. Peers within a neighborhood can exploit the heterogeneity their processing capabilities by engaging in the specialization pattern, so that each neighborhood of same-type peers coalesces and stabilizes around one or more hyphal peers of the same type.

The clustering dynamics in Mycoload thus provide additional self-organized logic, which extends, with considerations of neighbor types, the basic topology dynamics that regulate the construction of the superpeer overlay, as per feature (B). That additional logic underlying the Mycoload implementation of COLLECTIVE SORT induces peers that have not managed to achieve a target number of same-type neighbors to explore their environment by querying neighboring superpeers for other same-type peers; the superpeers act as matchmakers, attempting to connect their own neighbors with same-type peers that are one link removed. Notice that any peers that are unable to find matches using these mechanisms also engage in random exploration, using the peer list obtained through GOSSIP as per Feature (A).

The specific rules used by Mycoload to implement its version of COLLECTIVE SORT are the versions of *graph_movement*, *graph_pick_up*, and *graph_drop* discussed in Section V-B.

(D) Load Balancing

Since all the responsibility for building, maintaining and adapting the overlay is handled by features (A), (B) and (C), the self-organized load balancing can rely on the properties of the resulting overlay. Load-balancing has thus simply the responsibility of equalizing levels of work among peers offering the same type of service, scaled according to each peer's capacity.

This is a natural match for the GRADIENT pattern. The jobs in the queue of a peer are considered analogous to the quantity of some chemical marker, and the peer capacity equates to a volume in which the chemical is present. A peer can easily determine the relative concentration of the chemical marker relative to its neighbors, and, based on this information, can determine whether some number of jobs need to be transferred in order to achieve a balanced load. As a result, excess jobs diffuse through the system, with more jobs proportionally accumulating at higher-capacity peers.

GRADIENT is implemented once again through the basic patterns of SPREADING and AGGREGATION. For the SPREADING portion, peers have access to information about their direct neighbors via the overlay network, including their capacity and queue length. Peers periodically randomly select a neighbor with which to balance queues, and the peer with the higher relative concentration transfers the job to the other. AGGREGATION is performed when a peer receives jobs from a neighbor and they are merged into its own queue, ordering them such that each receives fair service based on its time of entry and other features.

VII. DISCUSSION AND CONCLUSIONS

We have presented a case study showing the pattern-based reverse-engineering of an entire self-organized software system, Mycoload. This is something that, to our knowledge, has not yet been documented elsewhere in the literature on self-organized software, whose focus tends to be the identification of a single design pattern that characterizes the primary behavior of some self-organized system, or collecting previously identified patterns. This paper addresses the next stage of this problem: isolating patterns from within an existing, complex distributed system that relies on multiple self-organizing mechanisms, and using those patterns to derive a modularized design.

The case study also demonstrates the effectiveness of a well-organized catalog of design patterns [6] for the engineering of self-organizing software. An outstanding challenge in this field is understanding the global emergent behavior that results from multiple interacting mechanisms. A pattern-based design – as shown in this case study – provides leverage for this problem, since the resulting system decomposition highlights the interactions among patterns, and their contribution to the overall self-organizing dynamics.

We see these as significant steps toward a methodology for engineering self-organizing software systems.

A benefit of a pattern-based approach is the possibility of identifying additional self-organizing behaviors within the system, which can be abstracted out, and can themselves become reusable mechanisms. In our case study, we have been able to identify two such modules, the SPECIALIZATION and COLLECTIVE SORT patterns. Both of these capture mechanisms that are present in a number of other self-organized software systems: SPECIALIZATION solves a the recurring problem of self-selected differentiation of roles and responsibilities among system elements or agents; and COLLECTIVE SORT enables the organization of disparate kinds of data or agents into homogeneous groups.

Another benefit for designers of self-organized software is that—once they can reuse and compose well-understood building blocks—they can focus more on the high-level dynamics, and how those dynamics are affected by adjusting the rules and parameters characterizing those lower-level blocks. Since it is well known that, in systems with emergent behavior, simple changes can have a large impact on the overall dynamics, that benefit can support the maintenance and evolution of the system. For example, our immediate future work on Mycoload is a version that can handle clustering and load balancing for multi-service peers—as opposed to peers that host only one service. For that, we plan to leverage our reverse-engineered design model, which highlights the logic that currently implements the collective sort functionality, to analyze what changes need to be made to it, and—critically—how those changes may impact or be impacted by the dependencies and interactions with other self-organizing mechanisms that the design model makes explicit. Moreover, since most of those other mechanisms are modularized as patterns, that is likely to help us limit the scope of parameter exploration, tuning and testing that is a necessary step any time new or modified self-organizing functionality must be implemented.

As the number of self-organization design patterns that are extracted from existing systems increases, and as we make progress in understanding how they relate and how they can be used together, the process we have demonstrated in this paper holds a great promise to provide the engineers of self-organized software with increased insight on the principles around which self-organization mechanisms can be built repeatably and leveraged effectively for the self-adaptation of large- and ultra-large scale systems.

ACKNOWLEDGEMENTS

The authors would like to give special thanks to Daniel J. Dubois, Elisabetta Di Nitto, and Nicolò M. Calcavecchia for their contributions to the development of Mycoload and to Rachel Greenstadt for her contributions to the development of Myconet.

REFERENCES

- [1] F. Heylighen, "The science of self-organization and adaptivity," *The Encyclopedia of Life Support Systems*, vol. 5, no. 3, pp. 253–280, 2001.
- [2] F. Dressler and O. B. Akan, "A survey on bio-inspired networking," *Computer Networks*, vol. 54, no. 6, pp. 881–900, 2010.
- [3] H. Parunak and S. Brueckner, "Software engineering for self-organizing systems," in *Proc. of 12th Int'l Wkshp. on Agent-Oriented Software Engineering (AOSE 2011)*, 2011.
- [4] L. Gardelli, M. Viroli, and A. Omicini, "Design patterns for self-organizing multiagent systems," in *2nd International Workshop on Engineering Emergence in Decentralised Autonomous System (EEDAS) 2007*, T. D. Wolf, F. Saffre, and R. Anthony, Eds. ICAC 2007, Jacksonville, Florida, USA: CMS Press, University of Greenwich, London, UK, June 2007, pp. 62–71.
- [5] T. De Wolf and T. Holvoet, "Design patterns for decentralised coordination in self-organising emergent systems," in *Proc. of 4th Int'l Conf. on Engineering Self-Org. Systems*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 28–49.
- [6] J. L. Fernandez-Marquez, G. Di Marzo Serungendo, S. Montagna, M. Viroli, and J. L. Arcos, "Description and composition of bio-inspired design patterns: a complete overview," *Natural Computing*, pp. 1–25, 2012.
- [7] G. Valetto, P. L. Snyder, D. J. Dubois, E. D. Nitto, and N. M. Calcavecchia, "A self-organized load-balancing algorithm for overlay-based decentralized service networks," in *Proceeding of SASO'11*. IEEE, 2011, pp. 168–177.
- [8] P. Snyder, R. Greenstadt, and G. Valetto, "Myconet: A fungi-inspired model for superpeer-based peer-to-peer overlay topologies," in *SASO'09*, 2009, pp. 40–50.
- [9] R. Nagpal, "A catalog of biologically-inspired primitives for engineering self-organization," in *Engineering Self-Organising Systems, Nature-Inspired Approaches to Software Engineering. LNCS Vol. 2977*. Springer, 2004, pp. 53–62.
- [10] M. Mamei, R. Menezes, R. Tolksdorf, and F. Zambonelli, "Case studies for self-organization in computer science," *Jrnl. of Systems Architecture*, vol. 52, pp. 443–460, Aug. 2006.
- [11] H. Kasinger, B. Bauer, and J. Denzinger, "Design pattern for self-organizing emergent systems based on digital infochemicals," in *Proc. of the Int. Conf. on Engineering of Autonomic and Autonomous Systems (EASe'2009)*. IEEE Computer Society, 2009, pp. 45–55.
- [12] H. Parunak, S. Brueckner, D. Weyns, T. Holvoet, and P. Valckenaers, "E pluribus unum: Polyagent and delegate mas architectures," in *Proc. of 8th Intll Workshop on Multi-Agent-Based Simulation (MABS07)*. Springer, 2007, pp. 36–51.
- [13] M. H. Cruz Torres, T. Van Beers, and T. Holvoet, "(no) more design patterns for multi-agent systems," in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOPES'11, NEAT'11, & VMIL'11, ser. SPLASH '11 Workshops*. New York, NY, USA: ACM, 2011, pp. 213–220.
- [14] J. Sudeikat and W. Renz, "Engineering environment-mediated multi-agent systems," D. Weyns, S. A. Brueckner, and Y. Demazeau, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Toward Systemic MAS Development: Enforcing Decentralized Self-organization by Composition and Refinement of Archetype Dynamics, pp. 39–57.
- [15] O. Babaoglu, G. Carright, A. Deutsch, G. A. D. Caro, F. Ducatelle, L. M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, and T. Urnes, "Design patterns from biology for distributed computing," *ACM Trans. on Autonomus and Adaptive Sys.*, vol. 1, pp. 26–66, 2006.
- [16] A. J. Ramirez and B. H. C. Cheng, "Design patterns for developing dynamically adaptive systems," pp. 49–58, 2010.
- [17] G. Chen and D. Kotz, "Solar: A pervasive-computing infrastructure for context-aware mobile applications," Department of Computer Science, Dartmouth College Hanover, NH, USA 03755, Tech. Rep., 2002.
- [18] E. Di Nitto, D. J. Dubois, R. Mirandola, F. Saffre, and R. Tateson, "Applying self-aggregation to load balancing: experimental results," in *BIONETICS'08*. ICST, 2008, pp. 14:1–14:8.
- [19] G. Nitschke, M. Schut, and A. Eiben, "Emergent specialization in biologically inspired collective behavior systems," in *Intelligent Complex Adaptive Systems*, A. Yang and Y. Shan, Eds. IGI Publishing, 2008, pp. 215–253.
- [20] N. Salazar, J. A. Rodriguez-Aguilar, and J. L. Arcos, "Robust coordination in large convention spaces," *AI Communications*, vol. 23, no. 4, pp. 357–372, 2010.
- [21] P. Kersch, R. Szabo, Z. Kis, M. Erdei, and B. Kovács, "Self organizing ambient control space: an ambient network architecture for dynamic network interconnection," in *Proc. of 1st ACM Wkshp. on Dynamic Interconnection of Networks*. ACM, 2005, pp. 17–21.
- [22] J. L. Fernandez-Marquez, J. L. Arcos, and G. D. M. Serungendo, "A decentralized approach for detecting dynamically changing diffuse event sources in noisy WSN environments," *Applied Artificial Intelligence*, vol. 26, no. 4, pp. 376–397, 2012.
- [23] R. Nagpal, "Programmable self-assembly using biologically-inspired multiagent control," in *1st Int'l. Joint Conf. on Autonomous Agents and Multiagent Systems: Part 1*, 2002, pp. 418–425.
- [24] H. Mühleisen, A. Augustin, T. Walther, M. Harasic, K. Teymourian, and R. Tolksdorf, "A self-organized semantic storage service," in *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2010, pp. 357–364.
- [25] T. Wang and H. Zhang, "Collective sorting with multiple robots," in *Robotics and Biomimetics, 2004. ROBIO 2004. IEEE Int'l Conf. on*. IEEE, 2004, pp. 716–720.
- [26] J. Sudeikat and W. Renz, "Toward systemic mas development: Enforcing decentralized self-organization by composition and refinement of archetype dynamics," *Engineering Environment-Mediated Multi-Agent Systems*, pp. 39–57, 2008.
- [27] S. Zhongzhi *et al.*, "A clustering algorithm based on swarm intelligence," in *Proc of Info-tech and Info-net, 2001. ICII 2001-Beijing.*, vol. 3. IEEE, 2001, pp. 58–66.
- [28] S. Selvakenedy, S. Sinnappan, and Y. Shang, "A biologically-inspired clustering protocol for wireless sensor networks," *Computer Communications*, vol. 30, no. 14-15, pp. 2786–2801, 2007.
- [29] L. Gardelli, M. Viroli, M. Casadei, and A. Omicini, "Designing self-organising MAS environments: the collective sort case," *Environments for Multi-Agent Systems III*, pp. 254–271, 2007.
- [30] M. Casadei, L. Gardelli, and M. Viroli, "Simulating emergent properties of coordination in maude: the collective sort case," *Electronic Notes in Theoretical Computer Science*, vol. 175, no. 2, pp. 59–80, 2007.
- [31] M. Casadei, M. Viroli, and L. Gardelli, "On the collective sort problem for distributed tuple spaces," *Science of Computer Programming*, vol. 74, no. 9, pp. 702–722, 2009.
- [32] J. Deneubourg, S. Goss, N. Franks, A. Sendova-Franks, C. Detrain, and L. Chrétien, "The dynamics of collective sorting robot-like ants and ant-like robots," in *Proc. of 1st Int'l Conf on Simulation of Adaptive Behavior on From Animals to Animats*, 1991, pp. 356–363.
- [33] M. Jelasity, W. Kowalczyk, and M. Van Steen, "Newscast computing," Vrije Universiteit Amsterdam Department of Computer Science Internal Report IR-CS-006, 2003.