

A Pattern-Based Architectural Style for Self-Organizing Software Systems

Jose Luis Fernandez-Marquez,
Giovanna Di Marzo Serugendo
University of Geneva
Centre Universitaire d'Informatique
Carouge, SWITZERLAND

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

Abstract—We present an architectural style for self-organizing software systems, which leverages a catalog of design patterns for self-organization. The catalog, which represent our prior work, defines a number of bio-inspired self-organization mechanism as design patterns, and shows how more abstract and complex forms of self-organization can be composed from simpler, lower-level mechanisms. We now discuss how the design patterns in our catalog can be situated and accommodated within an architectural style that uses them as building blocks to support the engineering of self-organizing software systems. Our style, which has a general layered organization, highlights the role of an Environment layer, which provisions several elementary and atomic self-organization mechanisms, which we call Core primitives. Those are leveraged by more sophisticated self-organizing mechanisms, which are requested on demand as Services by application-level Agents to effect their own individual self-organization behaviors. According to this layered organization, a full-fledged self-organized application activates agents with the appropriate behaviors, and parameterizes those behaviors according to its overarching requirements. We fully describe this architectural style, highlighting its merits and limitations, and discuss a possible implementation.

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

I. INTRODUCTION

With the continuous increase in runtime scale and complexity of software systems, self-adaptivity has assumed a central role in the software engineering discourse, and has been often mentioned as one of the defining challenges for the discipline [1], [2], [3].

Self-adaptivity is the ability of a software system or application to automatically modify its structure and behavior at runtime in order to ensure, maintain or recover some functional or extra-functional properties, even in the face of unexpected changes to operating conditions or user requirements. This kind of resilience to change is a typical property of the software architecture; moreover, self-adaptive provisions are often in charge of properties and end-to-end quality factors of the software (such as availability, reliability, response time and performance, robustness, etc.) that are architectural in nature. It is therefore quite natural to approach the engineering of self-adaptive software from an architectural perspective, and a large body of research has

indeed addressed this topic.

Architecture-centric approaches to self-adaptation rely upon an explicit representation of the system at runtime (its architectural configuration), obtained through monitoring and reflection; they reason about that configuration and its properties, and enact adaptations deemed necessary or useful by manipulating the configuration or architectural elements (either components or connectors). This approach assumes a high degree of accuracy and completeness in the information about the runtime architectural layout, and global, top-down control on the makeup of that layout and its adaptation.

Although this approach has been successfully used in many cases, there are some forces, in a number of systems and domains, that may test them severely. A major force of this type is uncertainty, with respect to either the collection of information about the runtime system, or the ability to carry out its adaptation as planned [4]. A domain in which high levels of uncertainty are intrinsic is that of systems that are strongly decentralized and have open boundaries. By open boundaries we mean that the number or even the type of participating elements in the system may vary at any given time, making impossible to maintain an accurate snapshot of the architectural configuration, or other global knowledge; rather, every individual element may hold its own partial view of the system, together with other local information that may be useful for adaptation. By decentralized we mean that no one element can take far-reaching control actions that influence most or all of the other elements in the system; rather, every individual element may only take local action that affects itself and possibly a limited number of other system constituents (*e.g.*, its neighbors).

Systems with these characteristics are becoming common as the increase in scale, complexity and pervasiveness of information technologies accelerates, inducing real-world examples of ultra-large-scale software systems [5], such as peer-to-peer networks, ubiquitous computing, and transport control and energy distribution systems.

Our interest focuses on this kind of system, because, in order to enable self-adaptivity, researchers have resorted to *self-organization principles and algorithms* [6], [7], [8], [9], often derived from biological or otherwise natural phenomena that can be observed in Complex Adaptive Systems

(CAS) [10]. Software self-organization can be regarded as a fundamentally different kind of self-adaptation, in which control is exerted in a bottom-up, emergent fashion, by a plurality of individual system elements that take actions with only local effects and local interactions.

While several self-organization mechanisms have been identified and have proven effective in producing system-wide adaptation in a number of applications, it is commonly recognized that it is quite difficult to represent and reason about this type of adaptation. For example, it is typically extremely hard to capture the local-to-global linkage [11], that is, how local behaviors in a particular system combine up to produce the desired global effects, or indicate under what set of conditions that can successfully happen. That, in turn, hinders the disciplined re-use of these mechanisms, and the engineering of self-organizing software in a repeatable way with predictable outcomes. The ability to frame self-organizing software systems within an architectural setting would therefore be an important advancement, as underlined also by the Self-Organized Architecture (SOAR) workshops¹ recently held at conferences on self-adaptive systems (ICAC)² and software architecture (WICSA)³.

In this paper, we present a pattern-based approach to this problem. It proceeds from the observation that, in self-organizing software, a desired system-wide adaptation is often achieved through the combination of a number of self-organization mechanisms that operate at different levels of abstraction. Therefore, our vision is twofold. First, we need to organize existing, well-understood mechanisms as a set of building blocks that can be composed together. To that end, we have developed a catalog of design patterns that capture self-organization mechanisms that recur in many existing systems [12], [13], [14]. The catalog systematizes these patterns by identifying the interrelationships that exist among them, including the composition of simpler, lower-layer patterns in more abstract, higher-layer ones. Second, we need an architectural style in which those patterns can be implemented and can operate as modular primitives or operators, in ways that are explicitly specified and produce predictable effects. In this paper, we discuss for the first time such an architectural style, which we have devised on the basis of the insight gained by using the patterns in a piecemeal fashion. We discuss the merits and limitations of this style, and also present some alternatives, in which the patterns play an analogous role.

The rest of the paper is organized as follows: we discuss the relevant literature in Section II, and our own pattern catalog in Section III. We introduce our architectural style in Section IV and an example of its use in Section V. We present some possible technological incarnations of the

style in Section VI. We conclude with a discussion of the properties of this approach and outline the direction of our future work.

II. RELATED WORK

The literature on architectural approaches to developing self-adaptive software is vast. A thorough overview is beyond the scope of this paper, and several recent surveys on the state of the art of software engineering for self-adaptive systems devote substantial attention to architectural aspects, challenges and techniques [2], [15], [3].

Here, we simply point out a few seminal papers, and how they have outlined a very common model for architectural adaptation, based upon the use of an adaptation engine, that is, an architecture evolution manager that operates at run time [16]. A classic example is found in Rainbow [17], which uses a centralized adaptation engine that reasons about an up-to-date representation of the runtime target system, and uses primitives that manipulate its architectural configuration based on quality considerations. Georgiadis *et al.* [18] propose a multiply instantiated adaptation engine together with the individual elements of the architecture. That idea goes in the direction of distributed—but not decentralized—control, because, to correctly enact adaptation, the distributed engine still relies on a complete and consistent view of the target architecture across all instances. Building on that work, Magee and Kramer [19] also elaborated a layered architectural reference model for self-adaptive systems that organizes distributed adaptation in a hierarchical fashion: elements within a higher layer work at higher level of abstraction, and drive the layer below. Although made of physically distributed and functionally distinct elements, an adaptation engine built in accord to this reference model is conceptually a single entity that still exerts centralized control on a target system.

Although this kind of approach can be quite successful, it has recognized limitations. From [2]: “... *the centralized control loop pattern ... may suffer from scalability problems. There is a pressing need for decentralized, but still manageable, efficient, and predictable techniques for constructing self-adaptive software systems*”. Two years later, in 2011, [3] states: “*While promising work is emerging in decentralized control of self-adaptive software there is a dearth of practical and effective techniques to build systems in this fashion*”, which we take as a lack of engineering, and, more specifically, architectural understanding of how existing decentralized control mechanisms effectively lead to self-organization in a predictable and repeatable fashion.

Since our contribution with this paper tries to address exactly that issue, we focus below on design patterns and architectures for self-organizing systems, and how they have been used to attack it.

¹<http://distrinet.cs.kuleuven.be/events/soar/>

²<http://nscac.rutgers.edu/conferences/icac/>

³<http://www.wicsa.net/>

A. Design Patterns for Self-Organizing Software

In the last decade, self-organizing mechanisms have been applied in different domains, achieving results that may go beyond what is possible with more traditional self-adaptation approaches, as documented for example in the SASO conference series⁴. However, researchers usually apply these mechanisms in an ad-hoc manner, preventing these mechanisms from being systematically reused to solve recurrent problems. Thus, the issue of systematizing those self-organization techniques lends itself to a pattern-based approach.

As with other design patterns, the main promise of self-organization patterns is to support the understanding of the correct scope of a particular solution, in terms of what problems it can successfully address, and under which set of conditions. Abstracting the problems and the solutions is particularly important, since the same self-organization technique can be at times used in very different application contexts, and—conversely—very similar problems can often be addressed via several diverse techniques.

Among the works that attempt to define design patterns for self-organizing software, some focus on the discovery and definition of a single pattern [20], [21]. Others propose a catalog of multiple patterns [22], [9], [6], [7]. Our previous work [12], discussed in Section III, is most similar to the latter. One thing that sets apart our catalog of bio-inspired patterns from previous efforts our organization of the patterns into layers, and we have drawn and documented composition relationships among them. That helps differentiate the patterns and scope them correctly. As pointed out by Parunak and Brueckner [23], the definition of composition and decomposition relationships is quite important, because—while it seems clear that certain self-organization mechanisms can be obtained from finer-grained—which primitives to use and how they should be combined is often not clear, and almost never explicitly codified.

B. Architectural Approaches to Self-organization

While self-adaptive architectures have been discussed extensively in the literature, self-organization is less frequently addressed directly, and even then it is often as an adjunct consideration. Parunak and Brueckner [23] extensively examine a number of important issues related to the software engineering of self-organization, including architectural concerns. One of the key challenges that architectures for self-organizing software needs to address is composability of multiple self-organization mechanisms in a single system. In that direction, Custa and Romey [24] present a taxonomy of self-adaptive architectural elements in a distributed context, and an algebraic approach to analyzing composition of adaptive elements in such a system. Specific architectural proposals are primarily in the Multi Agent System (MAS)

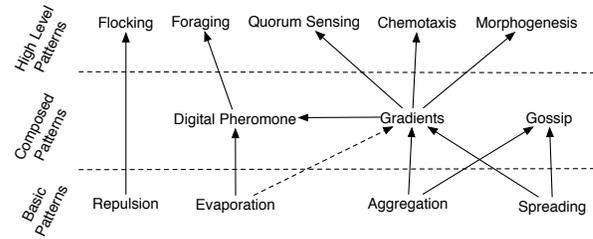


Figure 1: Patterns and their Relationships

vein, as in Wenkster *et al.* [25], which proposes a two-level agent model, with higher-level agents acting as explicit controllers, or SodekoSV [26], which pairs agents with a coordination medium hosted on an agent middleware.

In previous work, we proposed MetaSelf, a preliminary description of an architecture that combines both self-management through the enforcement of pre-defined system-wide policies, and self-organizing, through rules locally applied by entities [27]. That proposal, though, does not focus on the use of self-organizing mechanisms as primitives provided by the environment. More recently, we proposed BIO-CORE [28], a middleware built around a data repository and chemical rules that also exploit low-level self-organizing mechanisms as basic primitives provided by the computational environment. Although a precursor of the architectural style proposed in this paper, BIO-CORE is not explicitly described as such.

III. DESIGN PATTERN CATALOG

To facilitate the systematic engineering of self-organizing software systems, in our previous work we have constructed a catalog of bio-inspired modular and reusable design patterns, organized into layers. We report here the main characteristics of each of these patterns (*i.e.*, the problem they solve and the solution they provide). For a full description of each of these patterns (including class diagrams, sequence diagrams and examples of use) the reader should refer to [14], [13], [12].

By analyzing the behavior and properties of bio-inspired mechanism applied to self-organization of software that recur in the literature, we have isolated several mechanisms, which are basic and atomic, plus others that are composed from the basic ones. As a result, we classified our patterns into three layers. At the bottom layer are the *basic* mechanisms that can be used individually or in composition to form more complex patterns. At the middle layer there are the mechanisms formed by *combinations* of the bottom layer mechanisms. The top layer contains even more complex patterns that capture more complex and abstract bio-inspired self-organization mechanisms, and show different ways to *exploit* the basic and composed mechanisms from the bottom and middle layer.

⁴<http://www.saso-conference.org/>

Figure 1 shows the different design patterns collected in the catalog and their relations. The arrows indicate how the patterns are composed. A dashed arrow indicates that the use of the pattern is optional. One important observation about this classification is that Basic and Composed patterns capture behaviors that correspond to processes that occur within the environment in biological systems. For example, SPREADING, AGGREGATION and EVAPORATION are processes that change the environment in which self-organizing agents are immersed. Moreover, these environmental mechanisms become essential building blocks for more complex behaviors on the part of those agents.

A good and well-known example is ant foraging. In ant colonies, the ants are able to achieve very complex self-organized behaviors; this coordination is performed via pheromones, where ants create, follow, and reinforce pheromone gradients that form trails through the environment. This mechanism, to be effective, assumes an environment in which pheromones are spread (allowing ants to sense pheromone trails), aggregated (increasing pheromone concentration and reinforcing certain trails), and evaporated (allowing ants to adapt to changes as old trails that are no longer relevant are forgotten).

In line with the importance accorded by Weyns *et al.* to the concept of agent's environment [29], we consider the two lower layers, that is, the Basic and Composed patterns, as primitives for bio-inspired self-organization algorithms embedded within the execution environment. The High-Level patterns capture instead more complex processes that are the prerogative of each individual agent, and can be implemented either within the agent itself, or as services that agents can invoke. This insight is the basis for the software architecture proposed in the next section that integrates these design patterns in a modular fashion, hence promoting their reuse.

IV. ARCHITECTURAL STYLE

An architectural style defines the common architectural traits of a family of software systems by describing a structural organization. Architectural styles can be used on their own or combined together to form new architectural styles.

Following Garlan and Shaw [30], we will discuss our proposed architectural style for Self-Organizing Systems in the following terms: we will describe a structural organization for the architecture (including Components, Connectors, Data, Constraints on these entities and their Relationships); we will indicate Invariants of the components or connectors; we will also discuss Examples, Advantages and Disadvantages, and the underlying Computational Model.

A. An Architectural Style for Self-Organizing Systems

As observed in [31], the main design elements of a self-organizing system are: the *environment* in which the system

evolves; The *agents*, autonomous, individual, active entities of the system; the *self-organizing mechanisms* governing the behavior of the agents and their interactions with environment or other agents; and the *artifacts*, which are the passive data entities maintained by the environment, and created, modified and/or sensed by the agents (*e.g.*, digital pheromones spread in the environment or information exchanged among agents).

The main idea behind the architectural style we propose here is the provision by the environment of low-level, atomic self-organizing mechanisms under the form of *Core Services* or primitives, which are leveraged by more sophisticated self-organizing mechanisms, or *Services*, which in turn are requested on demand by *Agents* acting in the system to effect their own, complex self-organizing behaviors. It is then possible to build self-organizing applications using these agents, which activate individual behaviors with the purpose of creating a convergence to the kind of emergent order that would satisfy the application requirements.

We want such an architecture to support reuse (*e.g.*, a primitive such as evaporation provided by the environment can be used independently by multiple different *Services*, and activated by all agents that make use of those *Services*) and separation of concerns (*e.g.*, the programmer of the agent or the application concentrates on the functionality to provide, and relies on the underlying self-organizing mechanisms provided by the environment).

B. Structural Description

The architectural style we propose for self-organizing systems is a combination of three other architectural styles: Layered, Blackboard and Implicit Invocation.

Our architectural style follows overall a **Layered** organization (see Figure 2). At the bottom layer, we find the Computational Environment, in which a set of *Core Services* are executed and provided as primitives. Those primitives correspond to Basic or Composed patterns in our design pattern catalog. This is a parallel with the natural environment in which many biological self-organization processes take place, in particular with respect to production and distribution of information. Therefore, the data that represent the properties of the Computational Environment and the information deposited in it is managed and manipulated in a shared repository. Zooming into the Computational Environment layer, we see that it is composed of: (1) *Core Services*, that enact low-level self-organizing mechanisms corresponding to the Basic or Composed patterns of Figure 1 (see Section III), such as SPREADING, AGGREGATION, and EVAPORATION; and (2) the *Blackboard component*, which hosts the shared repository of data among Agents and Services, and is responsible to trigger the execution of Core Services as necessary.

In the higher layers of our architecture, we have *Services* and *Agents*.

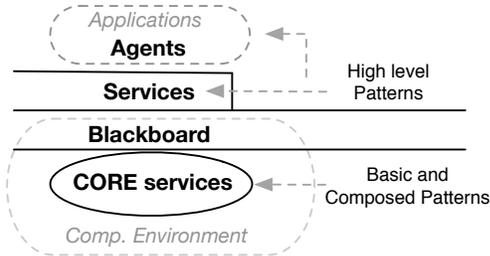


Figure 2: Overall Layered Architecture

Services are an abstraction that enables us to situate and enact complex self-organizing mechanisms, corresponding to the high-level patterns in our catalog, such as CHEMOTAXIS or QUORUM Sensing (see section III), which Agents make use of. A major difference between a Service and a Core Service resides in the fact that a Core Service is of an atomic nature and triggered by the environment, while a Service represents a composed self-organization mechanism that may also need some additional logic to be performed. Also, unlike Core Services, Services are not necessarily available at all locations where a Blackboard is instantiated.

Agents stand for any application-related computational component which runs some self-organization algorithm as its own individual behavior. Agents may implement some of the high-level patterns themselves, but – for the sake of reuse – they may even simply delegate them to Services and invoke them as needed.

Both Services and Agents can make use of the Core Services provided by the Computational Environment by interacting with the Blackboard, but they can also interact with one another.

Through the injection of appropriate data that Agents or Services trigger the activation of Core Services by the Blackboard. Notice that, unlike traditional Blackboard architectures, the Blackboard component we consider is not centralized for the whole system, but multiple Blackboard components are distributed through the system. Thus, each computing node which hosts some Services or Agents will also host one Blackboard component. Agents and Service may only access the Blackboard running on the same computing node, which thus restricts them to use local information only. However, Core Services, such as SPREADING, which are present in all Blackboards, provide a way for information to travel in controlled ways, and, effectively, connect the different Blackboards. (see Figure 3).

The self-organization primitives mechanisms representing Core Services are "embedded" within the Blackboard by means of the **Implicit Invocation** architecture (Figure 4). Upon arrival of appropriate data in a Blackboard repository, the corresponding Core Services are activated by that Blackboard component.

The distributed Blackboard component of our style thus allows the coordination between different software agents, even though they belong to different applications or they are

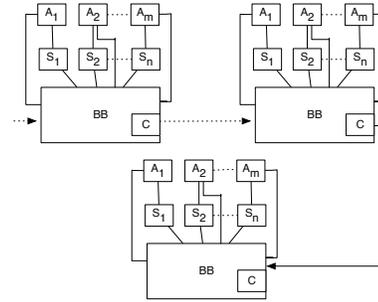


Figure 3: Blackboard Architecture

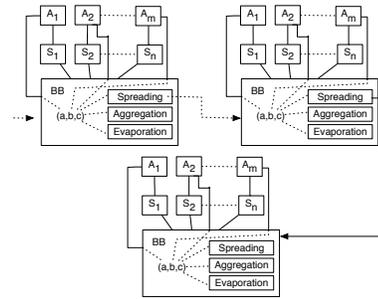


Figure 4: Implicit Invocation of Core Services

hosted in different nodes. Moreover, the Blackboard provides an elegant way for implementing indirect communication, which is a key communication model for nature-inspired self-organizing systems such as those based on ant foraging, gradient propagation or chemotaxis.

C. Components and Connectors

Following the description above, the *Components* of the architectural style for self-organizing systems are: the Agents, the Services enacting high-level self-organizing mechanisms, the Blackboard (repository and computational component) and the Core Services enacting basic self-organizing mechanisms.

There are different *Connectors*, as shown in figure 5:

- Between Agents/Services and Blackboard repository: data inserted/read/removed from the Blackboard repository;
- Between Agents and Services: data transiting in the Blackboard repository, changing its state, observed by either Agents or Services, possibly following a specific protocol;
- Between Agents/Services and Core Services: implicit invocation by the Blackboard component upon an event generated by the arrival of an appropriate data in the Blackboard repository;
- Between Agents/Services and remote Services: The SPREADING Core Service acts as connector for implicit invoking Services that are connected to remote Blackboards.
- Between Blackboards and Core Services: the Blackboard component activates the appropriate Core Service

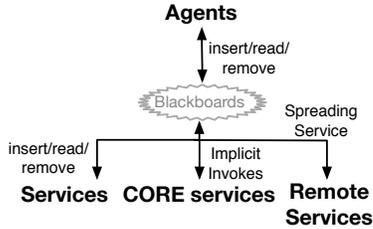


Figure 5: Components and Connectors

as a result of the corresponding event arising in the system (e.g. the arrival of a data in the Blackboard repository requesting to be spread, evaporated, etc.).

Data inserted, read and retrieved from the Blackboard repository consists of tuples of data with specific tags recognized by other Agents, Services or acting as events for triggering Core Services.

D. Invariants

The invariants of a software architecture refer to those features that should remain constant for different implementations. The invariants of our proposed architecture are as follows:

- Nodes participating in the system have only local connections, and the Blackboard component stored in each host is only directly accessible by Agents and Services running in the same node.
- The Blackboard component should provide at least three primitives: (1) Injection of data, (2) Observation of data, and (3) Removal of data.
- The invocation of Core Services is done by specific tags (e.g. keywords) present in the data injected into the Blackboard repository.
- The invocation of high-level Services can be done by tags in the Data, or by using a service description. In both cases the invocation is done by injecting some Data into the Blackboard repository.
- Agents and Services announce themselves in the system by injecting suitable data into a blackboard. These data will contain an Agent or Service description or information relative to the application or Service they provide.
- The order of executions of invoked Services cannot be established at design time. However, Core Services could have priority since they will be provided together with the Blackboard.
- Core Services are available at all nodes through the Blackboard components. In contrast, high-level Services may not be available at all nodes.

E. Advantages

Implicit invocation provides strong support for reuse of low-level self-organizing mechanisms. Additional low-level self-organizing mechanisms can be introduced in the system by simply registering them for the appropriate events.

Services can be published on the Blackboard and spread or made available at remote locations using the SPREADING Core Service. Moreover, Services could also be published using gradients (i.e., a GRADIENT Service), providing routes and information about distance (i.e., number of hops) to potential users (i.e., higher-level Services or Agents). Thus, the application can adapt dynamically, taking into account the current Services available in the system.

The composition of implicit invocation and Blackboard provides a way to publish Services at remote nodes. Applications, can thus be composed of Services, even though those Services are not hosted in the same node. This type of composition allows the system to increase available functionality, replace a Service in case of failure, or if there is a decrease in the quality of Service, to provide or find a replacement and hence a robust system.

Our approach presents scalability in two sense: (1) Applications can easily increase their functionality in a modular way, and (2) Both Services and Core Services proposed as a basis for all the applications developed following this architecture are very scalable, since they are based only on local interactions, and partial knowledge of the system. Their scalability has been properly demonstrated in many application as discussed in the coordination and self-organization systems literature.

F. Disadvantages

The main disadvantage of this architecture is that high-level components such as Agents or Services cannot rely on any order of computation in which the Core Services are invoked. Agents and Services cannot rely on a pre-defined Core Services flow of control.

G. Underlying Computational Model

The computational model is fully described in [28]. An abstract view of a setup for an individual computing host or node that complies with our description of the style is shown in Figure 6(a) a concrete realization, including the interfaces and main interaction channels between the various architectural elements found in a host, is shown in Figure 6(b). Agents are autonomous and proactive software entities running in a host; they can be programs in a traditional computing environments, but could also be, e.g., mobile robots or smart phones with sensors and actuators for control of its physical environment. Our architecture is distributed on as set of connected computing nodes (or Hosts) that host Agents, the Blackboard and Services.

V. CASE STUDY

We present hereby the implementation of a self-organizing algorithm in accord to our architecture. For illustration purposes, the example is kept very simple, and merely computer the sum of a distributed collection of numbers, with the data spread among multiple hosts. The purpose

Primitive Vocabulary	Informal Description
<i>Components:</i>	
Agent	Autonomous software component, usually pertaining to an Application - can be mobile or permanently hosted in one node.
Service/Remote Service	Software component enacting a high-level self-organizing mechanism - invoked by Agents possibly remotely.
Core Service	Software component enacting a basic self-organizing mechanism - invoked by Agents or Services.
Blackboard	Components hosting a shared repository where data is injected, observed and modified by Agents, Services and Core Services. Triggers Core Services when necessary.
<i>Connectors:</i>	
Agents/Services - Blackboard Repository	Data inserted/read/removed from the Blackboard repository.
Agents - Services	Exchange of data through the Blackboard repository.
Agents/Services - Core Services	Generation of an event arising from the insertion of a data in the Blackboard repository
Agents/Services - Remote Services	Exchange of data through Blackboard repositories, use of Spreading Core Services to propagate data at remote nodes.
Blackboards - Core Services	Invocation of Core Services in response to the arrival of expected data.

Table I: Primitive Vocabulary

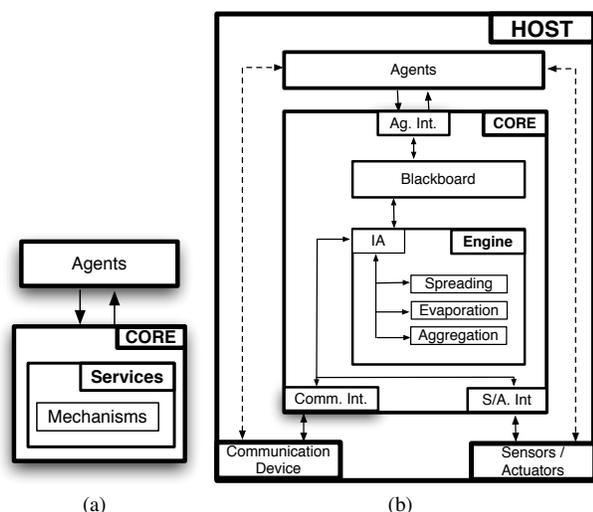


Figure 6: System's Structure

of the example is to show how multiple self-organization mechanisms in our pattern catalog are accommodated and executed as needed within the proposed architecture.

A. Description

Let N be a set of nodes with local connections to other nodes. Each node contains a middleware instance that provides a Blackboard and the Core Services of SPREADING, AGGREGATION, and EVAPORATION. As discussed in Section IV, these services will be implicitly invoked based on the properties of the data injected into the Blackboard. The middleware also provides an execution environment for Services and Agents.

Every node hosts an instance of a GRADIENT Service gs . Upon invocation, this service creates a Data element that (a) will SPREAD from node to node, increasing its hop count by one every time it spreads to a new node; that will AGGREGATE the neighboring hop-count values to determine the lowest; and that will EVAPORATE, removing it from the

system after period of time. The Service then publishes this element to the local Blackboard.

Each node also hosts a CHEMOTAXIS Service cs , which monitors the local Blackboard for data elements associated with a GRADIENT. If such an element is found, it determines the neighbor with the lowest hop-count value and modifies it so that the SPREADING Core Service will transfer it to the appropriate neighbor Blackboard.

Each node $n_i \in N$ has an instance of an Agent a_i . This Agent stores an integer value v_i , which is a random number $[0..1000]$ (7(a)). The process is initiated by a single node n_{start} , and this node will contain the final sum.

(1) The initiating Agent a_{start} (running on node n_{start}) makes a request to the GRADIENT Service gs to establish a new gradient for this execution of the sum algorithm.

(2) The GRADIENT Service gs formats a data element representing a request that a gradient g be created, and publishes the request to the Blackboard.

(3) The Core Services running on the Blackboard spread the gradient throughout the network.

(4) Once the gradient is established at a node n_i , (Figure 7(b)), the node's Agent a_i pushes a data element to the blackboard containing the value of v_i , with flags indicating that it is associated with the gradient g , and that it should aggregate with other data elements of the same type.

(5) When the CHEMOTAXIS Service cs observes a data element associated with g at the node, it determines the lowest hop-count neighbor, and marks the data element to be transferred there.

(6) The SPREADING Core Service transfers the data element to the indicated neighbor. If multiple data elements associated with g are present, the AGGREGATION Core service will combine them by summing their associated values. (Figure 7(c)(d))

(7) When a data element associated with gradient g arrives at n_{start} 's Blackboard, the agent a_{start} removes it and adds the contained value to its local total. The algorithm converges when all the values have reached this node.

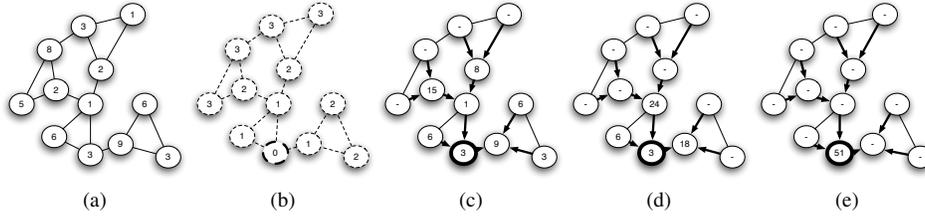


Figure 7: Sum algorithm

(Figure 7(e))

VI. IMPLEMENTATION FRAMEWORK

The proposed architectural style could be implemented in a number of different ways. In this section we discuss one possible approach that has been developed as part of the SAPERE European Project.⁵ This approach is a reification of the vision of the architectural style that is discussed in Section IV, and provides a framework for the style, which can thus support the development of decentralized, self-organizing applications from the pattern primitives discussed in Section III.

We identify the main components and attending decisions that have been adopted within the SAPERE framework. For a more extensive discussion of this framework, readers are referred to [32]. In the SAPERE implementation the main concepts are:

- “Live Semantic Annotation” (LSA): An LSA is a tuple that represents any information about an agent or service. LSAs are injected in the LSA’s space representing the (updated) state of its associated component. An LSA represents the data described in the proposed architecture for self-organizing systems.
- LSA’s Space: a distributed, shared space where context and information are provided by a set of LSAs stored at given locations. According to our proposed architecture, the LSA’s Space corresponds to the Blackboard component and its repository.
- ECO-LAWS: Eco-laws are rules that implement Core Services. These rules act on the LSAs stored in each LSA’s Space by deleting, updating or moving LSAs between LSA’s Spaces.
- LSA Bonding: An LSA bond acts as a reference to another LSA and provides fine-tuned control of what is visible or modifiable to each Agent and what is not. If an LSA of a given Agent includes a bond to an LSA of another Agent, the former Agent can inspect the state and interface of that other Agent (through the bond) and act accordingly.
- Agents: Agents execute in hosts and are able to locally access the LSA’s Space available in that host.

⁵www.sapere-project.eu

Property	Description
ID	Unique identifier
EVAP	Activate the Evaporation Service. Automatically this property sets up the Relevance attribute (REL)
AGGREGATE	Activate the AGGREGATION Service
SPREAD	Activate the SPREADING Service
DATA	Actual information stored in the LSA

Table II: LSA’s Properties

ECO-LAWS reside in the LSA’s Space and they are invoked following an implicit invocation pattern.

As a proof of concept, preliminary results are shown in [32], where a crowd steering application is simulated.

That simulation clarifies the concepts, developed in terms of a set of eco-laws of general applicability, addressing basic mechanisms of field diffusion, gradient establishment, and LSA bonding.

To give the flavor of a translation into the SAPERE framework of the proposed architectural style, we show here the SPREADING and AGGREGATION Core Services and use of them by another simple self-organized application: Reaching an Agreement, which has been presented in full in [14].

An LSA can be subject to different eco-laws depending on the specified properties or tags of the tuple. Table II shows the properties that one LSA should contain to trigger the Core Services.

1) *SPREADING Core Service*: The SPREADING Service retains a copy of the information received or held by an agent and sends this information to the neighbors, so that it propagates over the network. The eco-law corresponding to the SPREADING Service is as follows (1):

$$\langle \text{ID}, \text{SPREAD}, \text{DATA} \rangle \xrightarrow{r_{spr}} \langle \text{ID}, \text{DATA} \rangle, \text{bcast}(\text{ID}, \text{SPREAD}, \text{DATA}) \quad (1)$$

When an LSA with SPREAD property set to true is processed, the LSA is identified as a spreading LSA (i.e. the LSA is subject to the SPREADING eco-law (1)). The LSA is sent to the neighboring LSA’s spaces (with a broadcast *bcast*) and a local copy of the information remains. This process is repeated by neighboring LSA’s spaces, causing the LSA to be propagated over the whole system, making the information contained in the LSA available to all the

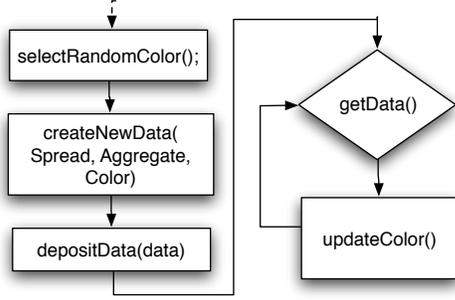


Figure 8: Reaching an agreement: Flow Diagram

Agents participating in the system. All Core Services are currently described with default rates.

2) *AGGREGATION Core Service*: The AGGREGATION Core Service consists in locally applying a fusion operator to synthesize information. When an LSA with the AGGREGATE property equals true is processed, the LSA is identified as an aggregation LSA (i.e. the LSA is subject to the aggregation eco-law (2)). The AGGREGATION Core Service retrieves all the LSAs that are going to be aggregated. These LSAs are those with the aggregation property equals to true and that share the same application identifier (provided by the field APP_{ID}).

The eco-law for the AGGREGATION Service is as follows:

$$\begin{aligned}
 &\langle ID_1, AGGREGATE, APP_{ID}, DATA_1 \rangle, \dots, \\
 &\langle ID_n, AGGREGATE, APP_{ID}, DATA_n \rangle \xrightarrow{r_{agg}} \\
 &\langle ID_{new}, AGGREGATE, APP_{ID}, Avg(DATA_1, \dots, DATA_n) \rangle
 \end{aligned} \quad (2)$$

The above eco-law for the AGGREGATION Service assumes the average as a default operator (Avg).

3) *Reaching an Agreement*: This application is derived from GOSSIP, a composed mechanism, where the SPREADING and the AGGREGATION mechanisms are used simultaneously. In the gossip process the information is sent over the network using the SPREADING mechanism and it is aggregated at each node with the local information by using the AGGREGATION mechanism. In this example, the GOSSIP enacted by an application’s Agent in each node actively uses the SPREADING and AGGREGATION Core Services.

In this application the system has to reach an agreement on the nodes’ color. Initially, each Agent at each node has a random color and during the simulation the color of the nodes must converge to a common color.

Figure 8 shows the flow diagram for each agent in the Reaching an Agreement application. The different steps are presented as follows: (1) each agent in each node initially chooses a color at random; (2) it creates an LSA, with properties SPREAD, AGGREGATE set to true and the color as DATA (see LSA (3)), indicating the application id APP_{ID} ; (3) the LSA is injected into the LSA’s space; (4) SPREADING and AGGREGATION Core Services then act on these LSAs: the SPREADING Core Service broadcasts LSAs to all neighboring nodes. Inside each node, the AGGREGATION Core Service then acts on all LSAs whose AGGREGATE property is

set to true. These LSAs are provided by the neighbors under the effect of the SPREADING Core Service. AGGREGATION averages the color data (only one piece of data per node will remain, containing the average color of the node and that of its neighbors); (5) agents periodically read their local LSA’s space, and retrieve the new aggregated LSA; (6) each agent then updates its color to the one contained in the DATA field provided by the retrieved LSA and returns to step 5.

$$\langle id, SPREAD, AGGREGATE, APP_{ID}, DATA = nodeColor \rangle \quad (3)$$

VII. CONCLUSIONS

In this paper, we have presented an architectural approach suitable to pattern-based engineering of self-organizing software systems. Our contribution comprises a catalog of reusable design patterns capturing mechanisms for self-organizing dynamics, and a composite architectural style that supports the composition and reuse of such mechanisms into full self-organized applications. This architectural style has a layered organization. A key aspect is represented by the Computational Environment layer, where Core Services reside to enact basic self-organizing mechanisms that describe processes of information management and manipulation. This way, the architecture clearly separates the responsibility of the computational environment from that of higher-level Services and application-level Agents, which make use of the processes in the Computational Environment level to implement more complex individual behaviors. This style can facilitate the design and implementation of many self-organizing applications or services, and fosters reuse of well-understood self-organizing mechanisms that recur in the literature in a unified way. The use of Blackboards and Implicit Invocation within our composite style facilitates service composition and easily supports the entry and exit of Services and application Agents to and from the system. Overall, it enables a dynamic infrastructure composed of a potentially very large number of nodes, and is applicable to application domains such as Mobile Ad-hoc Networks, P2P, Wireless Sensor Networks, ubiquitous computing, and ecosystem of services.

Our next steps will entail further validation of this architectural style in practice. In particular, we intend to assess how well it facilitates the use and reuse of self-organizing patterns all the way up to the Agent level, and how effectively it eases the development of self-organizing applications comprised of many distinct agents with addition logic and individual parameterization.

REFERENCES

- [1] P. Horn, “Autonomic computing: Ibm’s perspective on the state of information technology,” *Computing Systems*, vol. 15, no. Jan, pp. 1–40, 2001.
- [2] B. C. et al., “Software engineering for self-adaptive systems: A research roadmap,” in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Berlin / Heidelberg, 2009, vol. 5525, pp. 1–26.

- [3] R. de Lemos et al., "Software Engineering for Self-Adaptive Systems: A second Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Dagstuhl Seminar Proceedings, R. de Lemos, H. Giese, H. Müller, and M. Shaw, Eds., no. 10431. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011.
- [4] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming uncertainty in self-adaptive software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 234–244. [Online]. Available: <http://doi.acm.org/10.1145/2025113.2025147>
- [5] L. Northrop, P. Feiler, R. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan et al., "Ultra-large-scale systems: The software challenge of the future," *Software Engineering Institute*, 2006.
- [6] T. De Wolf and T. Holvoet, "Design patterns for decentralised coordination in self-organising emergent systems," in *Proceedings of the 4th international conference on Engineering self-organising systems*, ser. ESOA'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 28–49. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1763581.1763585>
- [7] O. Babaoglu, G. Canright, 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 Autonomous and Adaptive Sys.*, vol. 1, pp. 26–66, 2006.
- [8] M. Mamei, R. Menezes, R. Tolksdorf, and F. Zambonelli, "Case studies for self-organization in computer science," *Journal of Systems Architecture*, vol. 52, pp. 443–460, August 2006. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1163824.1163826>
- [9] 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.
- [10] F. Heylighen, "The science of self-organization and adaptivity," *The Encyclopedia of Life Support Systems*, vol. 5, no. 3, pp. 253–280, 2001.
- [11] T. De Wolf and T. Holvoet, "Towards a methodology for engineering self-organising emergent systems," 2005. [Online]. Available: <http://www.cs.kuleuven.be/~textasciitilde/tomdw/publications/pdfs/2005soas.pdf>
- [12] J. L. Fernandez-Marquez, G. Di Marzo Serugendo, S. Montagna, M. Viroli, and J. L. Arcos, "Description and composition of bio-inspired design patterns: a complete overview," *Natural Computing Journal (to appear) (Avail. at http://www.cui.unige.ch/~dimarzo/paperstmp/NACO2011.pdf)*, 2012.
- [13] J. L. Fernandez-Marquez, J. L. Arcos, G. Di Marzo Serugendo, M. Viroli, and M. Sara, "Description and composition of bio-inspired design patterns: The gradient case," in *Workshop on Bio-Inspired and Self-* Algorithms for Distributed Systems (BADS'2011)*. ACM, 2011, pp. 25–32.
- [14] 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 *Proc. of the Int. Conf. on Engineering of Autonomic and Autonomous Systems (EASE'2011)*. IEEE Computer Society, 2011, pp. 87–96.
- [15] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.
- [16] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," in *Proceedings of the first workshop on Self-healing systems*, ser. WOSS '02. New York, NY, USA: ACM, 2002, pp. 21–26. [Online]. Available: <http://doi.acm.org/10.1145/582128.582133>
- [17] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.
- [18] I. Georgiadis, J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems," in *Proc. of 1st Wkshp. on Self-Healing Systems*. ACM, 2002, pp. 33–38.
- [19] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE'07*. IEEE, 2007, pp. 259–268.
- [20] 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.
- [21] H. Parunak, S. Brueckner, D. Weyns, T. Holvoet, and P. Valckenaers, "E pluribus unum: Polyagent and delegate mas architectures," in *Proc. of 8th Intl Workshop on Multi-Agent-Based Simulation (MABS07)*. Springer, 2007, pp. 36–51.
- [22] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85029-8_4
- [23] H. Parunak and S. Brueckner, "Software engineering for self-organizing systems," in *Proc. of the Twelfth International Workshop on Agent-Oriented Software Engineering (AOSE 2011)*, 2011.
- [24] C. Cuesta and P. Romay, "Elements of self-adaptive architectures*," in *Proceedings of the Workshop on Self-Organizing Architecture (SOAR 2009)*. Springer, Heidelberg, 2009.
- [25] R. Wenkstern, T. Steel, and G. Leask, "A self-organizing architecture for traffic management," *Self-Organizing Architectures*, 2009.
- [26] J. Sudeikat, L. Braubach, A. Pokahr, W. Renz, and W. Lamersdorf, "Systematically engineering self-organizing systems: The sodekovs approach," *Electronic Communications of the EASST*, vol. 17, no. 0, 2009.
- [27] G. Di Marzo Serugendo, J. Fitzgerald, and A. Romanovsky, "Metaself - an architecture and development method for dependable self-* systems," in *The 25th Symposium on Applied Computing (SAC 2010)*. Sion, Switzerland: ACM, 2010, pp. 457–461.
- [28] J. L. Fernandez-Marquez, G. Di Marzo Serugendo, and S. Montagna, "Bio-core: Bio-inspired self-organising mechanisms core," in *6th International ICST Conference on Bio-Inspired Models of Network, Information, and Computing Systems*. Available at <http://www.cui.unige.ch/~dimarzo/paperstmp/BIONETICS2011.pdf>, 2011.
- [29] D. Weyns, A. Omicini, and J. Odell, "Environment as a first class abstraction in multiagent systems," *Autonomous Agents and Multi-agent Systems*, vol. 14, no. 1, pp. 5–30, Feb 2007.
- [30] M. Shaw and D. Garlan, *Software Architecture - Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [31] G. Di Marzo Serugendo, "Robustness and dependability of self-organising systems - a safety engineering perspective," in *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, ser. LNCS, vol. 5873. Lyon, France: Springer, Berlin Heidelberg, 2009, pp. 254–268.
- [32] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson, "Pervasive ecosystems: a coordination model based on semantic chemistry," in *27th ACM Symp. on Applied Computing (SAC 2012)*, S. Ossowski, P. Lecca, C.-C. Hung, and J. Hong, Eds. Riva del Garda, TN, Italy: ACM, 26-30 March 2012.