A Methodology for Classifying Self-Organizing Software Systems

Klaus Herrmann  Matthias Werner  Gero Mühl

Berlin University of Technology
Institute for Telecommunication Systems
EN6, Einsteinufer 17, 10587 Berlin, Germany
Email: {klaus.herrmann|m.werner|g muehl}@acm.org

Abstract: The software industry is faced with the fast growing complexity of IT infrastructures. This makes manual administration increasingly difficult and appears to be the limiting factor in the general development of such infrastructures. This complexity crisis has stimulated many researchers to propose software systems that are allegedly self-organizing. However, often, this claim is only based on vague intuitions about self-organization and a proper classification is missing. From a scientific standpoint, this is questionable and undesirable since it adds confusion rather than clarity. In this article, a framework is proposed that enables researchers to classify their systems and to state clearly, verifiably, and reproducibly why and in which way they are self-organizing. This framework provides a definition of the properties of a self-organizing system, it defines a respective class of systems denoted as SO, and it offers a methodology for proving that a given system is belonging to SO or to its complement SO. Some case studies of well-known existing software systems are presented to show that the methodology is useful and practically applicable.

Keywords: self-organization, classification, adaptivity, structure.

1. Introduction

The software industry is faced with a fundamental problem: Integrated, networked, enterprise computing systems are getting increasingly complex. This complexity stems from the fact that the size, the heterogeneity, and the amount of interactions in such systems are growing fast. The resulting complexity is overwhelming the administrative personnel. As a result, systems are operated suboptimally or run at the edge of collapse. This has been termed the “complexity crisis” [17] in reference to the software crisis that set in about 40 years ago.

The above problem has been recognized as the major limiting factor in the further development of software (and hardware) infrastructures. Consequently, major companies have started initiatives to render such infrastructures self-organizing. Examples include IBM’s Autonomic Computing Initiative [16], Intel’s Proactive Computing [27], and Microsoft’s Dynaic Systems Initiative [8]. Self-organization is the key element of any autonomic system (AS). Such systems are required to function with no or minimal external intervention. Thus, they need to be able to organize themselves to adapt to changes in their environment without a notable degradation in performance or break-down. The concept of self-organization encompasses a set of subordinate concepts that are commonly called self-* properties: Self-optimization, self-configuration, self-healing, and self-protection are all abilities of a system to adapt without external control in some specific way to dynamic changes in their environment.

Self-organization is a concept that is pervasive in most naturally occurring complex systems. Numerous examples can be found in physical, biological, social, and economical systems whose components are living creatures, inanimate physical objects, or a mixture of both. These examples include the well-organized dynamics in heated fluids and the behavior of sand piles as well as the activities in a colony of social insects, the intelligence arising from the seemingly amorphous substrate of the brain, and the complex economic structures created by people selling and buying goods on a global market. Unfortunately, many of the concepts and mechanisms that are working in a self-organizing system have only been defined in a relatively fuzzy way, so far. Today, science is able to understand the behavior of many of these systems by identifying the underlying concepts and by explaining how they interact in each concrete system. However, the research community is still far away from a general model of self-organization that may be transformed into an engineering approach in a straight-forward way. Nevertheless, computer scientists invent new systems and label them as being self-organizing every day. In many cases, their claim remains vague and unclear. The definition of self-organization and the interpretation of the concrete system within the boundaries of such a definition are completely left to the reader. There is a plethora of definitions in the area of self-organizing systems formulated by biologists [7], physicists [2][3], psychiatrists [21][12], chemists [20], and complex system researchers [14], just to name a few. However, compared to the large number of recent publications about self-organizing software systems (SOSS), there are only very few definitions against which one can firmly evaluate whether these systems really do exhibit self-organization features. From a scientific standpoint, this is questionable and undesirable since it adds confusion rather than clarity.

In this article, a framework is proposed that enables researchers to classify their systems and to state clearly, verifiably, and reproducibly why and in which way they are self-organizing. This work aims at contributing to a common understanding of the concept of self-organization in computer science and at providing researchers with a tool that makes their work comparable to others. This should lay the foundation for more productive discussions about self-organizing
computer systems. Moreover, the ability to classify existing systems may offer a new empirical bottom-up approach towards an engineering methodology: Studying those software systems that one recognizes as being self-organizing may eventually lead to a software-specific generalization of their features and, consequently, to common design principles. Thus, in the same way as biologists study insects and build models for their behavior, computer scientists could study existing self-organizing software systems and build new design models. This approach would be radically different from the top-down approaches proposed thus far.

The rest of the article is organized as follows: In Section 2, an overview over the general concepts that are observable in naturally occurring self-organizing systems is given. A review of related work is presented in Section 3. Section 4 provides the definitions of the terms self-organization, adaptivity, structure, and decentralization that the presented model is based on. Furthermore, the article defines a methodology for describing a system in terms of its key elements such that it can be classified. In Section 5, this classification methodology is described and a definition of the class SO of self-organizing systems is given. Section 6 is dedicated to the problem of verifying a given classification. This is followed by a case study of a selection of different well-known software systems in Section 7. The classification of these systems is shortly discussed to show the power of the classification methodology. In Section 8, some concluding remarks are given.

2. General Concepts

Naturally occurring self-organizing systems possess many desirable features. Well-known examples are colonies of social insects (e.g., ants and termites) or the human brain. Such a system is typically composed of a large number of identical, or at least very similar, components. This creates a massive redundancy and, therefore, a high degree of robustness since the loss of a single component does not affect the overall system. Typically, the components of a self-organizing complex system are comparably simple. A single neuron and an individual ant do not possess a great complexity by themselves. Instead, the interaction between these components plays the central role. This interaction usually results in a feedback process by which useful structures are built very quickly (positive feedback), until the depletion of some necessary resource leads to a negative feedback effect that controls the growth and stabilizes the resulting structures. Due to this inherent non-linearity, a complex self-organizing system like an ant colony cannot be understood simply by decomposing it and by analyzing the constituting components. This classical reductionist approach that has been popular in science for hundreds of years fails because the properties of a complex system are only present if the system acts as a whole.

Self-organizing systems can create amazingly complex adaptive behavior at the global level by virtue of their interactions despite the simplicity of their constituting components. They achieve this without any centralized controlling entity (i.e., a leader) and without any blueprint or template of the global structure [7]. This is, of course, highly desirable also in modern computing infrastructures.

From the above description, three major properties can be identified that are of interest with respect to the classification of SOSS:

(a) They are adaptive. In the absence of an external controlling entity, a SOSS must be able to cope with any disturbances or major changes in the environmental conditions by itself.

(b) They structure themselves. The term “organization” in “self-organization” implies that some kind of structure is established, maintained, and adapted by the system itself [3]. This goes beyond a pure adaptation as it involves dependencies among the system’s components.

(c) They are decentralized. The control of the adaptive structuring process is distributed over the components of the system.

These three properties are necessary and sufficient for any software system to be called self-organizing. Thus, there is no need to care about the concrete mechanisms that lead to these properties. Whether a system uses some bio-inspired feedback process to self-organize or whether it resorts to more traditional computer science approaches is of no importance for the presented classification model. This means that the presented model is applicable to any software system and, thus, very general in nature. Moreover, the model assumes that the creation of structure is directly related to the the adaptation of the system: A SOSS creates structure in order to adapt and not purely for the sake of creating structure. While this may preclude certain systems, it seems to be a useful restriction for purposes targeted in this article.

In the following sections, an overall model for SOSS that in turn consists of models for the three central properties of a SOSS is proposed. Each of these models provides a method for proving that a given system possesses the respective property. Subsequently, a classification methodology is discussed that is based on the models of adaptivity, structuring, and decentralization. In addition a system class SO is introduced, containing all SOSSs that comply with the presented model.

3. Related Work

The model that is proposed in this article is the first attempt to provide a more formal method for classifying software systems in terms of their self-organization ability in a systematic way. Numerous authors have suggested definitions of self-organization and the related concept of emergence in the area of computing. However, the majority of these definitions are only informal [15][6][9][23]. The presented work also starts from such an informal definition (Definition 1) to establish an intuition. But in contrast to the aforementioned studies, models for the three central properties are stated that are far more formal and that provide systematic ways of testing whether a system has these properties.

The only prior attempt to define what a self-organizing system is in a formal way was made by George Lendaris in 1964 [18]. Lendaris’ definition is a very rigorous mathematical one. In essence, it demands that a self-organizing system must have an internal component that controls the system by setting parameters such that the system’s function approaches a prescribed target function. This resembles a definition of adaptive systems made by Lotfi Zadeh one year earlier [29]. One major shortcoming of Lendaris’ definition is the fact that it completely ignores that self-organizing systems inherently create, maintain, and adapt some form of internal structure. This sets them apart from adaptive systems. Moreover, the abstract nature of the definition renders the task of mapping a real-world
A system can be tested for emergence by measuring the surprise felt by an observer as he is confronted with the system’s specification and its actual behavior. This resembles the well-known Turing test. However, the results achieved via such a test cannot be objective. Boschetti et al. investigate different definitions of emergence and describe how they could lead to tools for detecting emergence in a systems. Their principle means of investigation are multi-agent systems. While this work points into some interesting directions, it does not provide a concrete methodology. Shalizi et al. attempt to quantify the self-organization of a system by measuring the “internally-generated increase in the statistical complexity” [24]. The authors propose to use the “amount of information required for optimal prediction of the systems dynamics” to measure this increase. Such a quantification would be an interesting result. However, a given degree of self-organization is rather unintuitive and hard to interpret. It does not give insight into the quality of the processes leading to the self-organization.

Approaches to the design of self-organizing systems are proposed by Wright et al. [28] and by Gershenson [10]. Wright et al. study a measure that can be applied to a system and that can serve as a feedback for an optimization process. This process can then adjust the system’s parameters until a self-organizing behavior is created. It has not been shown yet that the proposed measure is general enough to enable such an optimization for arbitrary systems. Gershenson proposes a very abstract methodology for designing and controlling self-organizing systems. However, as the author himself admits, this methodology is rather philosophical.

### 4. A Model of SOSS

The following basic definition of a SOSS extends the definition of self-organization in the context of biological systems given by Camazine et al. [7] by explicitly demanding that a SOSS shall be adaptive:

**Definition 1 (SOSS).** A self-organizing software system is an adaptive software system that adapts to its environment by changing its structure. This structure at the global level arises solely from numerous interactions among the lower-level components of the system. Moreover, the rules specifying the interactions among the components are executed using only local information, without reference to the global structure.

Apart from the properties that were already defined, an additional requirement is implicit to this definition: The structure shall arise solely from the interactions of the system’s components. This precludes systems in which the global structure is explicitly encoded into the components. For example, a hypothetical system may consists of a set of dots that may move on a plain. The structure that shall be established by the group of dots is a square shape on the plane. If each dot has a priori knowledge of the exact coordinate of its final position, then the structure is not produced by the system itself, but rather by the entity that configured the system prior to its start. No interaction is required at all in order to produce the square. This system would not be regarded as being self-organizing. Similarly, the structure must not be encoded into the environment.

### 4.1. Adaptivity

The intuitive understanding of adaptivity tells that an adaptive system changes to fit its environment. In the following, this intuition is more formally defined by adopting a definition of adaptivity given by Lotfi A. Zadeh in 1963 [29]. In this definition, Zadeh takes a black-box approach in that he simply states how an adaptive system should behave and not how this behavior should be achieved internally. This allows us to take any existing system and test whether it is adaptive or not.

Let \( S \) be the system, and let \( \{S_n\} \) be a family of all possible time-dependent input functions for \( S \), where \( \gamma \) specifies the concrete input function for \( S \). Let \( P(\gamma) \) be a function that measures the performance of \( S \) under the input function \( S_n \). One says that \( S \) performs acceptably well under \( S_n \), if \( P(\gamma) \) is in a prescribed class \( W \) of performance functions: \( P(\gamma) \in W \). \( W \) defines a criterion of acceptability for \( S \). For example, if \( P(\gamma) \) is real-valued, then \( W \) could be the set of all performance functions whose values exceed a prescribed number. Thus, one may think of \( P(\gamma) \) as a fitness function for the system \( S \) and \( W \) would be the set of all function that exceed a certain fitness threshold. Zadeh states that “a primary characteristic of an adaptive system is its ability to perform acceptably well in a changing and/or incompletely known environment”. He defines this environment by a relation \( \gamma \in \Gamma \), where \( \Gamma \) represents a subset of functions from \( \{S_n\} \) that \( S \) may be subjected to. Based on this terminology, Zadeh defines an adaptive system as follows:

**Definition 2 (Adaptivity).** A system \( S \) is adaptive with respect to \( \{S_n\} \) and \( W \) if it performs acceptably well (i.e., \( P(\gamma) \in W \)) with every source in the family \( \{S_n\} \), \( \gamma \in \Gamma \). More compactly, \( S \) is adaptive with respect to \( \Gamma \) and \( W \) if it maps \( \Gamma \) into \( W \).

Figure 1 depicts this model of adaptivity. A specific set of input functions is selected by \( \Gamma \), and the system \( S \) is subjected to this set. The system’s behavior is assumed to be measurable and indicates the fitness of the system under the given input. This behavior is evaluated through the performance function, and eventually the system’s performance is judged using the acceptability criterion \( W \).

To prove that a given system is adaptive, one needs to map it to this definition and show that it maps \( \Gamma \) into \( W \). First, the (environmental) parameters that comprise the input function have to be defined. For a heater with a thermostat, for example, this parameter would simply be the ambient temperature. In general, a single input may be also be vector-valued. \( \Gamma \) is a certain range of temperatures that is normally perceived. Then, the performance function has to be defined. For the heater, this would be the room temperature. The criterion of acceptability \( (W) \) would be the class of all temperature curves that end up...
within a certain range of the desired temperature. If one can show that the thermostat can produce a temperature curve in $W$ for any ambient temperature in $\Gamma$, then this system is adaptive.

It should be noted that every conceivable system can be mapped to the model of adaptivity and shown to be adaptive. This mapping is a rather mechanical procedure that is not restricted in any way. Thus, trivial mappings are possible, too. This problem will be discussed when the methodologies for the classification and the verification of such a classification are explained.

4.2. Structure

The structures created by self-organizing systems can be very diverse. They may be spatial, temporal, spatiotemporal, or functional in nature [12][4]. Thus, the concrete definition of structure may vary widely depending on the specific system. An abstract definition may be formulated as follows:

**Definition 3** (Structure). *Structure* is the property of a system by which it constrains the degrees of freedom of its components.

For example, an ant colony that is not currently building trails may be viewed as being unstructured (see left side of Fig. 2) in the sense that the individual ants are free to adopt any arbitrary movement pattern. This results in the absence of any visible structure since the ants essentially move randomly across the available space. However, if they form trails, the ants are largely constrained to the movement along the pheromone paths (see right side of Fig. 2). Thus, the trail structure constrains each ant in the way it behaves compared to the unstructured case. An individual ant’s movement becomes dependent (by virtue of the pheromone) on the movement patterns of its fellow ants. Ross Ashby, one of the pioneers of the research on self-organization, calls this necessary component of organization “conditionality” [3].

Fig. 1. Zadeh’s model of adaptivity.

![Diagram of Zadeh’s model of adaptivity.](Image)

How can this abstract notion of structure be captured such that one can prove that a given system is structured? The concept of *entropy*, known from thermodynamics and also from Shannon’s theory of information [25] is of great help here. Entropy is a measure for the degree of disorder in a system. One may also say that entropy measures the *uncertainty* about the state of a system. If a given system is free to be in any of its possible states, the entropy is at its maximum. For such a system, it cannot be predicted what its state will be at any point in time, and one would describe such a system as completely disordered and random. If, however, the system is by some means constrained to a subset of all possible states, the uncertainty about its state decreases and, conversely, the degree of order in the system increases. Thus, entropy can be used to find out whether the degrees of freedom of a system’s components are constrained and, thus, whether that system exhibits structure.

Shannon defines information entropy as follows [25]:

$$H(P) = -K \sum_{s \in S} P(s) \cdot \log P(s)$$  (1)

where $K$ is a constant that decides about the measurement unit. $S$ is the *state space* of the system and $s \in S$ is one possible state. $P(s)$ is the probability for the system being in state $s$ according to the probability distribution $P$. In order to apply Shannon’s entropy to a given system, one first needs to define the state space $S$ for that system. The approach presented here assumes that this space consists of a finite number of discrete states the system can be in. If the system consists of $n$ components of which each may have an individual state, then the overall state is the Cartesian product of the states of all components: $S = S_1 \times S_2 \times \ldots \times S_n$, and a single state is defined as $s \in S = (s_1, s_2, \ldots, s_n)$.

For the trail-building ant colony, the state space of an individual ant may be defined in two dimensions. The first represents the ant’s position $(p)$ on the plane. This position may be expressed as the index of a region that the ant resides in, assuming that the plane is divided into a number of such regions. The second dimension of the state space represents the ant’s heading $(h)$ and may be expressed as a number between 0 and 359 or, more coarsely, as an orientation (north, south, etc.). Thus, each ant $a_i$ has a state $s_i = (p_i, h_i)$ and system’s overall state consists of all $n$ ant states. As the trail structure is created, one can expect that the position as well as the heading of the ants becomes less uncertain (cf. Fig. 2). The system then resides in a smaller subspace of its state space and consequently the entropy measure reduces.

Thus, statistical entropy is a good means for detecting structure and the increase of structure in a system. Of course, the creation of structure that is to be revealed using the entropy measure must be captured by the definition of the state space. Similar to the model of adaptivity, the description takes a vital role in the detection of structure.

---

1This approach is also possible for a continuous state space.
4.3. Decentralized Control

The third necessary condition for an SOSS, besides the creation of structure and adaptivity, is that it does not use some central controller to achieve this. But why is this part of the definition necessary? Why can the system not have one sub-component that is central within the system and controls the organizational process?

Consider a system $S$ that can be decomposed into a subsystem $C$ and a set of subsystems $S' = S \setminus C$ such that $S'$ loses its ability to self-organize. Then, $C$ obviously functioned as a central controller within $S$, and the original definition of what comprises a self-organizing system would actually abuse the meaning of “self” in “self-organizing”. $S$ does not organize itself; it rather consists of a distinct subsystem $C$ that organizes the rest of the system $S'$. If the definition of a self-organizing system would include $C$, then it would lose its discriminating power completely since it would include any system that creates organization in one way or the other. One could always extend any system’s definition until some controller is inside to make it “self-organizing”.

In order to prove evidence that a given system is in the class of SOSSs, a more formal statement is required about what decentralization means. To define the term “decentralization”, first, one has to state what the definition of the term “system” is. This will actually not add anything new or surprising to the informal understanding of the term. It just clarifies things for the succeeding discussion:

**Definition 4** (Software System). A software system $S$ consists of a set of well-defined interacting components. $S$ is identified with the set of its components, and any subset of $S$ is called a subsystem. Furthermore, every software system $S$ has a well-defined function $f_S$.

A system’s components may be heterogeneous or homogeneous. In the given context, one may identify the system’s adaptation to the environment as its function. In general, the function of $S$ may be any useful computation. A software system may have several independent functions. Based on Definition 4, a central controller is defined as follows:

**Definition 5** (Central Controller). A central controller $C$ of a system $S$ with respect to $f_S$ is a subsystem of $S$ that controls the actions of the remaining subsystem $S' = S \setminus C$ such that $S$ is able to perform $f_S$ but $C$ is unable to perform $f_S$ in isolation (i.e., without $S'$).

The reader may object that $C$ may control the actions of $S'$ and still be able to perform $f_S$ on its own, without $S'$. Why does the definition not allow this case? If $C$ is able to perform $f_S$ in isolation, then one of the following two statements must hold:

(a) $C$ can be reduced further by removing (controlled) components from $C$ and adding them to $S'$. Thus, $C$ contains an equivalent smaller central controller that complies to the definition and, therefore, the original decomposition did not succeed in isolating the controller.

(b) If that is not the case, $C$ is simply a more condensed decentralized version of $S$. What is the contribution of $S'$ to the execution of $f_S$, then? $S'$ could simply be removed from $S$ without destroying its functionality. This implies that $S'$ is redundant in the definition of $S$, and one would prefer the functionally equivalent but simpler decentralized system $C$ instead of $S$.

The first case is indicative of an insufficient decomposition while the second shows that the system was not defined properly. The following Decomposition Theorem simply follows from Definition 5.

**Theorem 1** (Decomposition Theorem). If a software system $S$ possesses a central controller, then $S$ can be decomposed into two subsystems $C$ and $S' = S \setminus C$, such that neither $C$ nor $S'$ can perform the original function $f_S$ of $S$. This kind of decomposition is called a destructive decomposition.

A colloquial way of expressing the essence of Theorem 1 is the following: If a system is controlled centrally, then it has a central point of failure. Accordingly, one way of proving that a system works in a decentralized fashion, is to show that it has no central point of failure. Note that, in general, the reverse implication is not true.

In order to prove that a given system is decentralized, one needs to show that it is impossible to decompose it destructively. According to Theorem 1, this impossibility implies the absence of a central controller. Based on the description of the system, one may not necessarily have to investigate any possible decomposition. A thought experiment in which one iteratively removes one component at a time may suffice in many cases. After each hypothetical component removal, one has to show that the system still fulfills its function.

4.4. Global Knowledge

It is generally assumed that self-organizing systems produce global patterns from local interactions. That is, the components of the system do not use any blueprint, template, or recipe; nor do they have a leader amongst them [7]. While the occurrence of a leader corresponds to a centralization of control, blueprints, templates, and recipes have not been discussed yet. A blueprint is an abstract plan present in the components of the system, similar to the way in which buildings are constructed following the blueprints made by architects. A template is a physically existing prototype of a structure that may be (more or less) copied to reproduce the structure. Finally, a recipe is a step-wise behavior that is hard-coded into the system’s components. If each component follows this recipe, the desired structure is generated. Camazine et al. deal with these alternatives to self-organization in great detail in their book on *Self-Organization in Biological Systems* [7].

It is rather obvious that none of these three alternatives is coherent with the concept of self-organization. After all, each of the three means for building structure is in some way imposed on the system from the outside. Thus, the system does not create structure by itself. Instead, it gets explicit instructions for the construction process from outside the system.

4.5. System Model

Based on the above model of a self-organizing system, arbitrary systems can be evaluated. It is important to note, however, that whether one recognizes a given system as being self-organizing, is completely depending on how its key elements are defined [7][11][29][15]. The three central models of adaptivity, structure, and decentralization are all based on different aspects of a single underlying system model. Of course, the same system may be modeled differently, and none of these models is canonical. For example, the definition of the state space of an ant colony presented in Section 4.2 is a focus on
the position and the heading of the ants in order to detect a trail structure. Using their body temperature and their color as state attributes, would not have revealed this process. However, neither of the two state space definitions is more valid in any absolute sense.

Thus, in order to classify any given system as being self-organizing, it is vitally important that precise definitions of its key elements are provided. Moreover, the same system may appear to be outside the class of self-organizing system when defined differently. Of course, any such definition is subject to dispute, and there may be more than one valid or correct definition. There is a certain degree of common sense involved with the assessment of such a classification. Someone who is faced with this task, has to check whether the given definitions are sensible and useful or whether they are overly artificial in order to make a given system appear to be self-organizing. In the following, a list of system elements is given that need to be defined properly as a basis of a classification:

- **The system itself:** It must be clearly stated which components the system comprises of. These are the components that act and interact to create structure within the boundaries of the system in order to adapt.
- **The environment:** The exact border that separates the system from the environment needs to be defined. Moreover, the way in which the environment interacts with the system needs to be described. Note that, with respect to Definition 2 the environment then defines \( \{ S, \Gamma \} \).
- **The system's internal structure:** A precise definition of what comprises the system’s structure must be given. The notion of structure may vary widely for different systems. The definition chosen here should capture the structural aspect whose change causes the adaptation. For example, this may be related to the spatial distribution of the system’s components or to their interconnectivity. In preparation of an uncertainty analysis using statistical entropy, a definition of the systems’s state space is required.
- **Good versus bad structure:** The system needs to be described. Note that, with respect to Definition 2 the system exhibits this kind of structural adaptivity without a central controlling entity. This can be done by showing that a destructive decomposition according to Theorem 1 is not possible. In addition, it must be shown that the system does not use global knowledge.

If it is possible to complete these steps for a given system successfully, then the classification is complete.

**Definition 6** (Class of Self-Organizing Software Systems). **SO** is defined as the class of self-organizing software systems and **SO** as its complement. A system \( S \) is in **SO** under a description \( \mathcal{D} \) (denoted as \( S_{\mathcal{D}} \in \text{SO} \)), if \( S_{\mathcal{D}} \)

(a) complies with the definition of adaptivity (step 2),
(b) adapts by changing its structure (step 3),
(c) does not employ central control (step 4).

The preceding discussion implies that a system \( S \) may have two different descriptions \( \mathcal{D} \) and \( \mathcal{D}' \) such that \( S_{\mathcal{D}} \in \text{SO} \) and \( S_{\mathcal{D}'} \notin \text{SO} \). Therefore, the description is an essential part of definition 6. Note that \( S_{\mathcal{D}} \notin \text{SO} \) and \( S_{\mathcal{D}'} \in \text{SO} \) are equivalent statements.

6. **Verification Methodology**

The fact that there is seldomly a single canonical description of a given system may lead to descriptions being tailored solely for the purpose of proving the system self-organizing. Therefore, the normal process of validating scientific work becomes especially important in this context.

If a researcher presents a system \( S_{\mathcal{D}} \) and makes the case for its ability to self-organize, then the scientific community should research a number of questions in the attempt to confirm or dispute this thesis. The special focus here is on the validity of the system description chosen:

- Are all system elements defined properly in \( \mathcal{D} \)?
- Are these definitions consistent with each other?
- Does \( \mathcal{D} \) present a useful view on \( S \) that captures \( S \)'s essence, or is it rather artificial and unintuitive?

If these questions yield satisfactory answers, the latter three steps should be checked for validity. If, however, there is reasonable doubt concerning the validity of the system description \( \mathcal{D} \), then steps 2 to 3 may be formally correct but the overall classification is not valid. In other words, \( S_{\mathcal{D}} \) may be formally recognized as belonging to \( \text{SO} \), but this classification is based on an invalid description \( \mathcal{D} \) and is, thus, invalid itself. Furthermore, there may be a more useful definition \( \mathcal{D}' \) that is preferable and under which \( S_{\mathcal{D}'} \notin \text{SO} \). If such a \( \mathcal{D}' \) exists, then it should be specified as a part of any negative verification.

7. **Case Studies**

To show that the SOSS model and the classification methodology presented in this article can be applied in practice, some well-known software systems will be analyzed and classified in the following. This section also serves as a stimulus for discussion since it may state classification results that are disputable. Note that not all details of every step of the classifications are
provided due to spatial restrictions. Instead, sketches are presented to establish an intuitive understanding.

7.1. Routing Information Protocol – RIP

RIP [13] is a classical interior gateway protocol that establishes routing tables in network nodes by exchanging distance information messages among neighboring routers. Each router (also called a node hereafter) regularly sends a list of known nodes and the distance (hop count) to these nodes to its neighbor routers. When a router receives such a message, it updates its routing table by adding 1 to each of the received hop counts and including this information into the routing table. The next-hop router for a node \( v \) is set to the neighboring node with the lowest hop count for \( v \).

In the following, the key elements of this system are briefly described according to Section 4.5:

- **System**: The RIP system consists of the routing daemons. These daemons are running the protocol and maintain the routing tables.
- **Environment**: The environment that the system adapts to is defined by the network. RIP adapts to changes in the network topology. If it is reconfigured (i.e., nodes are added or removed), the routing tables are eventually updated to reflect these changes and restore the system’s ability to send packets via the shortest routes.
- **Structure**: The structure established by RIP is given by the distributed routing tables that define a spanning tree for the network.
- **Good vs bad structure**: A good structure is a global configuration of the routing tables that (i) enables each pair of routers to communicate, and (ii) does so using the shortest possible routes.

RIP was designed to adapt to changes in the network topology. Thus, one would define all possible changes (adding and removing nodes, rewiring, etc.) that may occur in a network as the system’s inputs. As a performance function \( P(\gamma) \), we define the average number of hops required to transport a packet from its source to its destination. Note that this definition is rather simplified as it uses the implicit assumption that packets are generated uniformly over the whole network. Furthermore, the possibility of broken routes also requires some refinement. However, \( P(\gamma) \) measures how efficiently and effectively the routing system controls the transport of packets through the network. The acceptability criterion \( \mathcal{W} \) is defined by regarding the optimal routing algorithm that always chooses the shortest route and is always able to route packets to their destination as long as the network is not partitioned. This optimal algorithm \( \mathcal{O} \) may be simulated off-line with a global view of the network and against topology changes that were recorded and replayed. This algorithm is also subjected to the performance function and provides a benchmark. Based on this benchmark, \( \mathcal{W} \) is defined as the class of all functions that eventually fall below a constant factor of the optimal algorithm’s performance. Let us assume that \( \mathcal{O} \) produces an average hop count of \( h \) per packet. Then RIP performs acceptably well if it handles any possible topology change and reduces the average hop count per packet to at most \( c-h \) for \( c \geq 1 \) in finite time. We call a system that is able to re-establish its performance after changes in the environment to within a constant fact \( c \) of the optimal performance \( c \)-adaptive.

Without having conducted the respective experiments, experience shows that RIP is adaptive in the aforementioned sense. It clearly achieves this adaptation by updating the routing tables on all nodes according to the environmental changes. Thus, it adapts by changing its structure. Moreover, no single routing daemon assumes the function of a central controller. Since they are all homogeneous, one may remove any subset of routers without removing the overall system’s ability to route packets. That is, assuming that such a removal preserves the network’s validity (no partitions, etc.). Therefore, a destructive decomposition is not possible which implies that the RIP is decentralized. Finally, the RIP system does not employ global knowledge. Instead, distance information is propagated in an epidemic way via multiple hops until it reaches all nodes. No component of the system ever has a global view on the whole system, nor does any component ever store the complete structure (i.e., the spanning tree). In information stored in each router is local in nature. It may concern nodes that are far away. However, the further away two nodes are, the more outdated their mutual knowledge is due to the hop-wise propagation of information.

The result of the classification is that, under the given description, the Routing Information Protocol is self-organizing. In the following, the above description is denoted as \( \mathcal{R} \). Thus, \( \text{RIP} \in \mathcal{R} \).

7.2. Open Shortest Path First – OSPF

The OSPF routing protocol [19] has replaced the older RIP as the most popular interior gateway protocol. In essence, OSPF assumes the same function as RIP. However, it does so in a rather different way. In each network that runs OSPF, one router is elected as the Designated Router (DR). The DR collects information on the complete topology of the network and builds a network graph. This graph is sent to all remaining routers via multicast. Each router uses this complete network graph to calculate the shortest paths to all other routers using Dijkstra’s algorithm. These paths are then used for routing.

The description \( \mathcal{R} \) that was developed for RIP is also adopted for OSPF. Thus, the description of the system, the environment, and the structure as well as the mapping to the model of adaptivity remain unchanged. Obviously, OSPF establishes a structure that is equivalent (in principle) to the structure of RIP. It also adapts in the same way as RIP, albeit supposedly better due to the difference in knowledge provided to the routing daemons.

However, OSPF is not a decentralized system. The existence of the DR and the fact that each router has global knowledge about the network graph violates the definition of a SOSS. Moreover, the interaction among the routers are also not local. They are provided with global information in a global communication effort. Based on these facts, a classification yields that OSPF in not self-organizing (\( \text{OSPF} \notin \mathcal{SO} \)) due to its centralized nature.

7.3. Transmission Control Protocol – Slow-Start

The Transmission Control Protocol that is used in the Internet provides several means for avoiding network congestion. One well-established example of such a mechanism is slow-start [1] and its numerous variants. With slow-start, both sides of a TCP connection independently increase the size of the congestion window (number of packets that may be sent without
having received a corresponding acknowledgment (ACK)) iteratively until timeouts occur for ACKs. First, the window is increased exponentially until a threshold size is reached and after that, it is increased linearly. When an ACK is not received, the algorithm divides the threshold size by two and starts the algorithm anew.

Obviously, the system consisting of all the slow-start algorithms running in the TCP processes is adaptive to the current traffic in the network (the system’s environment). There is also no centralized control entity because each TCP sender acts autonomously (with respect to the slow-start algorithm). However, it is hard to discover any global structure in this system. Even in a single TCP connection, the congestion window on both sides may be completely independent from each other as the traffic in both directions may traverse different routes with different traffic characteristics. Is there an observable structure that enforces constraints on the congestion window of individual TCP senders? Can a state space be designed accordingly to prove that the overall system resides in a smaller subspace thereof? The authors are not aware of any empirical study that may reveal a global structure among the set of all congestion window sizes in the Internet (or a larger portion thereof).

The authors claim that TCP slow-start in not self-organizing \(\text{TCPSS}_\tau \notin \text{SO}\) due to the lack of global structure. \(T\) denotes the system description sketched for \(\text{TCPSS}\).

### 7.4. Chord – Distributed Hash Tables (DHTs)

Peer-to-Peer (P2P) systems are large-scale distributed systems in which data items (music files, movies etc.) are stored in a completely decentralized fashion. The participants of such a system (called peers) access data directly on other peers who store it. This is enabled by special search mechanisms. Chord [26] is a prototype example of a P2P system that relies on Distributed Hash Tables to allocate data items to peers and to establish an effective and efficient search mechanism. Names of data items as well as peer IDs are hashed to produce keys from a well-defined keyspace. A distance function is applied to the resulting keys that assigns a distance to any pair of keys. Each data item is stored at the peer whose key is closest to the item’s key. Routing tables are maintained at the peers that route any request to the neighbor whose key is closest to the key indicated in the request. Thus, through a closeness gradient, requests eventually reach the location of the desired data.

For this analysis, the Chord system is defined as consisting of the Chord daemons and their data storages. The Chord system is able to adapt when new peers join. As this introduces a new peer key in the system, a number of data items may have to be re-located to a newly added peer, and the structure of the system has to be changed to adapt to this change. In this case, the allocation of data items to peers is defined as the system’s structure. The restructuring is managed in a decentralized way. One can conclude that Chord in adaptive, it adapts by changing its structure, and it is decentralized. Thus, under the given description \(C, \text{Chord}_C \in \text{SO}\).

### 7.5. Results

As can be seen in these case studies, the given definition of SOSS possesses discriminating power. Given a description of the respective system, one may argue why it is in \(SO\) or why this is not the case. Table 1 gives an overview over the results of the short study.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>adapt.</th>
<th>struct.</th>
<th>decentral.</th>
<th>(\in \text{SO})</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIP(_R)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>OSPF(_R)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>TCPSS(_\tau)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Chord(_C)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

It is important to note that the above analyses have always concentrated on distinct aspects of the respective system. For example, it is not claimed that TCP as such and under any circumstances is not self-organizing. Instead, using the basic model for describing a system, the analyses presents a specific view on each system and argues about that view. Regarding the complexity of those systems that are worth such an analysis, one inevitably has to reduce them to their relevant aspects. Each system may have numerous different aspects that may be analyzed in isolation to produce independent classification results. This is an important result: The ability to provide a system model in a standardized way provides us with the means to study self-organization features at a different level of granularity. The same system may be analyzed under different valid descriptions. And under some of them, it might be in \(SO\) while it is not in \(SO\) for other descriptions. This is not necessarily a contradiction since many systems consist of numerous subsystems that may internally have self-organization features. The system model enables the isolation and the individual classification of each such subsystem.

The systems that were subject to the case studies have purposefully been chosen even though they are not necessarily at the top of the list of self-organizing systems that are vividly discussed within the community. Of course, ant-based routing could also be analyzed instead of classical routing algorithms. However, it is the opinion of the authors that broadening the view in the discussion on the principles of self-organization is vital. The presented work supports the claim that many existing systems possess self-organization abilities even though they are not inspired by the obvious swarm approach. Many of these systems have been around for a long time, and one may learn a lot by analyzing them on the basis of the SOSS model presented in this article.

### 8. Conclusions

In this article, a practical methodology for classifying software systems in terms of their ability to self-organize is presented. By giving a useful and valid system description that captures the system’s essence, and by mapping this description to the definition of SOSS, researchers are enabled to provide firm evidence of their claim that a given system is self-organizing. The article provides several case studies for well-known software system from different application domains. These studies show that the presented model of SOSS and the associated classification methodology can be applied in practice. Furthermore, the SOSS model is useful as it possesses enough discriminating power to establish non-empty system classes \(SO\) and \(\text{SO}\). Finally, the studies show that the model enables researchers to isolate only the relevant aspects of any given system to investigate them in individually. Thus, a more fine-grained classification becomes possible.

The authors believe that the presented methodology can
lead to a more coherent view on the class of self-organizing software systems. Moreover, it enables discussions and insights on existing systems which may bring more clarity into the area of self-organization in computer science. Thus far, a common taxonomy together with a methodology for classification like the one proposed here is missing in this community. This has lead to the occurrence of a plethora of different systems that co-exist but that cannot really be compared with each other in a unified way, even though they supposedly belong the same class of self-organizing systems.

The authors postulate that the presented methodology will yield three system classes: systems that are in SO beyond any doubt, systems that are in SO beyond any doubt, and systems that cause dispute as to whether they are in SO or not. The latter class is the most interesting one because it will most likely fertilize the discussion on self-organizing software systems in general.

9. Discussion and Future Work

The model that was presented here is a notable step forward from the current state of the art that provides only informal models as a basis for a discussion of SOSS. However, it still contains a number of informal elements as there is still some fuzziness in the general understanding of self-organization. Any fuzziness in a definition is, of course, undesirable. The authors are constantly trying to improve the overall model to make it more formal and, thus, more precise. At the same time, a certain level of abstraction is targeted to keep the task of mapping a given system to it as simple as possible.

The fact that a classification is always relative to a description may appear to be a major limitation in our model. However, one can argue that this problem is inherent to any model that attempts to capture a wide variety of systems. A canonical mathematical description does not exist since something as complex as a software system may be viewed from different perspectives. The presented approach explicitly deals with this problem by requiring a proper description that has to contain a specific list of elements. Furthermore, the methodology includes a verification step to get more objective results. Ideally, any classification is verified by more than one person. This is a process that is similar to the review process for scientific publications.

Currently, the authors are classifying a certain number of existing systems in order to compile a catalog of SOSSs. This will serve as a basis for an empirical study of SOSSs to extract common features beyond those covered by the proposed SOSS model. The authors hope that, eventually, this will lead to new approaches to the design of SOSSs.

Acknowledgement

Part of this work was supported by Deutsche Telekom.

References


Author Bios

Klaus Herrmann studied Computer Science at the University of Frankfurt/Main, Germany. He received his Ph.D. from Berlin University of Technology. Since September 2006 he holds a PostDoc position at the University of Stuttgart. His research interests are distributed systems, mobile computing, and self-organizing software systems.

Matthias Werner received a master degree in Electrical Engineering and a Ph.D. in Computer Science from the Humboldt University of Berlin. His research interests focus on non-functional properties, especially fault tolerance, real-time, and security. He has worked with Daimler Chrysler Research, UT of Texas at Austin and Humboldt University of Berlin. Currently, he is an assistant professor at the Berlin University of Technology.

Gero Mühl is a postdoctoral researcher at the Berlin University of Technology. His research interests include middleware, event-based systems, selforganization, and mobile computing. He received a master degree in Computer Science and a master degree in Electrical Engineering from the FernUniversity in Hagen, and a Ph.D. degree in Computer Science from the Darmstadt University of Technology.