This is a pre print version of the following article:

Methodological Guidelines for Engineering Self-organization and Emergence / Noel, Victor; Zambonelli, Franco. - STAMPA. - 8998:(2015), pp. 355-378. [10.1007/978-3-319-16310-9_10]

Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

29/03/2024 12:04

(Article begins on next page)

Methodological Guidelines for Engineering Self-organization and Emergence

Victor Noël and Franco Zambonelli

University of Modena and Reggio Emilia, Italy

Abstract. The ASCENS project deals with the design and development of complex self-adaptive systems, where self-organization is one of the possible means by which to achieve self-adaptation. However, to support the development of self-organising systems, one has to extensively resituate their engineering from a software architectures and requirements point of view. In particular, in this chapter, we highlight the importance of the decomposition in components to go from the problem to the engineered solution. This leads us to explain and rationalise the following architectural strategy: designing by following the problem organisation. We discuss architectural advantages for development and documentation, and its coherence with existing methodological approaches to self-organisation, and we illustrate the approach with an example on the area of swarm robotics.

Keywords: Self-organization, software architecture, problem decomposition, swarm robotics

1 Introduction

Engineering complex software intensive systems made up of large ensembles of components, and make them autonomic, requires a number of innovative models and tools. As it has been deeply investigated in the context of the ASCENS project, and is extensively reported in this book, these may include: new approaches to requirements engineering (as discussed in [1]), new programming languages (see Chapter I.1 [39]), and new methodological guidelines (see Chapter III.1 [31]).

In this chapter, we focus on a specific – yet very critical – methodological problem related to the engineering of complex autonomic service ensembles. Although the inherent goal of any software engineering approach is that to achieve – by design – a specific predictable behavior of the system, the complexity of large scale ensembles can sometimes undermine the possibility to fully achieve such goal. Indeed, as the scale of a system grows, the presence of non-linear interactions between its components and the lack of central control can make the appearance of emergent behaviours at the system level [38].

In particular, those systems showing "organised complexity" are of interest: their emergent behaviour can't simply be understood using statistical tools and

it is the organisation of the elements that matters [46]. An important mechanism governing complex systems is self-organisation: the autonomous change of the elements organisation without external control [15]. With self-organisation and emergence, complex systems are known for self-adaptivity: they adapt their functioning as a response to internal and environmental changes [29].

Johnson [33] noted that the apparent paradox between emergence and engineering stems from the confusion between emergence approached in a predictive way (i.e., being equated to surprising and often undesirable behaviours of the system) and emergence approached as a construct [27] (i.e., being equated to the appearance of high-level behaviours resulting from low-level simpler rules). Here, embracing the second vision, we assume (without arguing for the correctness of this choice) that self-organisation is the principle followed to design the low-level rules that lead to emergence.

Multi-Agent Systems (MAS) is one field where self-organisation and emergence are studied and applied to engineer self-adaptive systems [16, 42]. In such systems, the various agents organise themselves in an autonomous and decentralised way to change how the functionality of the system is realised or even to change the functionality itself. Some aspects of the global behaviour are not explicitly pre-designed but emerge at runtime through this self-organising process in an endogenous and bottom-up way: the agents are unaware of the organisation as a whole [42].

Precisely, the general challenge tackled here is engineering self-adaptive self-organising complex systems that exist in and modify a complex context while meeting complex needs, in the continuation of [12] and [1]. Towards that goal, we propose to look at practical methodological guidelines to accompany their design. In the following, we use the term "Self-Organising MAS" (SOMAS) to denote such engineered system.

The paper presents two strongly interrelated contributions. First, it proposes a clear understanding, using a software architecture and requirements vocabulary, of what it means to engineer a SOMAS. From that, it highlights and explains the role that the decomposition design activity plays for such systems: it acts as a design bridge between the problem and the emergent behaviour. Second, it explains and rationalises an architectural design strategy stating that when designing a SOMAS, the problem organisation can be exploited for decomposition: it is an enabler for emergent behaviours reached through self-organisation to be adequate with respect to the problem. We also discuss implications for the development and documentation of SOMAS in general.

This strategy is not a method by itself but a complement to existing approaches and methods: one of our objectives is to defend these contributions so that they influence existing and future methodological approaches to self-organisation and emergence as well as applications.

They are illustrated with an example taken from swam robotics IV.2 [43] : bots exploring and securing victims in an unknown environment. This example was used to elicit and try-out the contributions but is not the focus of this paper. Nevertheless, it is fully implemented and we propose a short comparison with

two other distributed algorithms to show the validity of applying the strategy and illustrate the differences it implies.

In Section 2, we review self-organisation and emergence from a software architecture and requirements points of view, and underline the importance of the role played by decomposition in the design of SOMAS. In Section 3, we present rationalise and discuss the defended strategy. In Section 4, we discuss in more details the example from swarm robotics. In Section 5, we relate the contributions with existing works that can be used to engineer emergence. We conclude in Section 6.

2 Emergence, Engineering and Decomposition

In this section, by interpreting the concepts of self-organisation and emergence from an engineering, i.e., architectures and requirements, point of view, we underline two important things: first, these approaches answer requirements but also impose design constraints, and second, some requirements are answered by the human designer while others are emergently answered at runtime. These facts are of course not novel by themselves but are usually not explained with an architecture and requirements vocabulary. This leads us to conclude on the importance of the role that the decomposition plays in the design of SOMAS.

2.1 Self-Organisation and Emergence

Emergence is considered present in a system if from two different point of view on the system (hence it is an observer-relative property), the behaviour exhibited by the first one — called high-level or macro-level — is determined (formally "supervenes on" [36]) by the second one — called low-level or micro-level —, and if the first one is easier to understand than the second one [13]. In other words, we say that a macro-level behaviour emerges from the micro-level interactions, that this macro-level behaviour is not contained in some elements of the micro-level and that the macro-level behaviours we consider are those that are of interest to the engineer of the system.

Even though this definition of emergence does not include self-organisation, emergence most usually appears in self-organising systems. The main important specificities of these systems are that they change their organisation (and thus their behaviour) without external control [15]. Together, self-organisation and emergence imply a decentralisation of control inside the system [29]: without it, there can't be emergence as the elements of the micro-level behaviour would contain the macro-level behaviour.

Here, emergence is about the state of the system at a given moment while self-organisation is about the dynamics that enable to reach this state. There exist other understandings of these concepts (cf. [13, 15, 16, 33, 42]) but they all describe the same reality: the global behaviour is found at runtime and not predefined in the elements or in the way they are composed. Figure 1 illustrates well this principle: the emergent function results from the composition of the

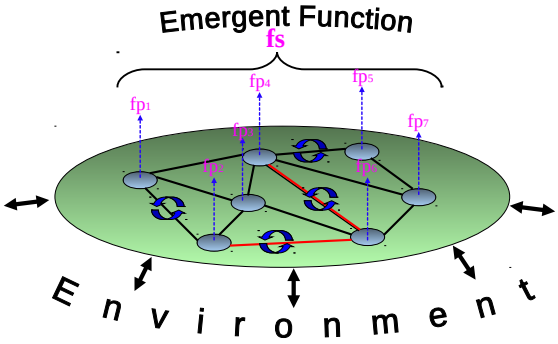


Fig. 1. Components and connectors view about the emergence of the global function from local functions through self-organisation [22].

local functions of the elements and can be changed by the latter by modifying their function or their relations.

2.2 Software Architecture, Problems and Requirements

In terms of software architecture, and in line with the general component-based approach of ASCENS (see Chapter I.2 [11]) the design activity of SOMAS focuses on producing components and connectors (C&C) views [5, 14] (i.e., the description of the runtime elements, their behaviours and their relations) of the solution to answer the requirements and the constraints of the problem tackled. The components are the agents, their environment connects them together and the relations between agents and with the environment are often dynamically changing at runtime [3]. Figure 1 is thus a SOMAS C&C view. Another important aspect of software architectures concerns modules views (i.e., the description of units of implementation and their relations): while the design of SOMAS does not usually directly cover them [3], as we will see in Section 3.6 the choices made during their engineering still impact them.

Problems of Interest What we call a problem to answer here is made of a context and requirements [28]: engineering is about finding the software solution, here the SOMAS, satisfying the requirements in that context.

As an example, in a robotics scenario (illustrated in Figure 2), we look at the search and secure problem: bots must explore an unknown environment to look for victims and then secure them (supposedly to rescue them, but this is not covered in our example). The context is composed of the bots (controlled by the software to build) with limited communication capabilities, the environment that have walls, the victims that must each be secured by several bots. The requirements are to search and secure victims, to secure them all, as fast as possible, to explore the accessible space fairly, to completely explore the space in a non-random way, etc.

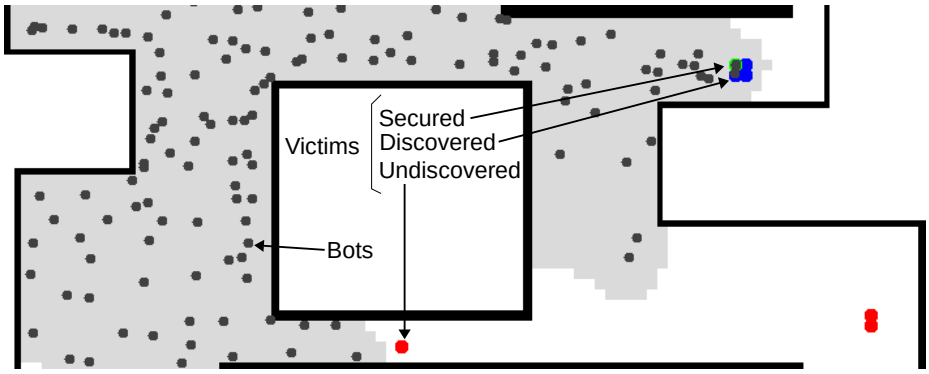


Fig. 2. Robotic scenario with bots, victims and environment to explore.

Some aspects of the problem won't change in any of its instances (e.g., bots capabilities), while some will be different (e.g., walls and victims positions, bots communication distance) and may even change at runtime (e.g., failure of bots). The type of SOMAS we want to be able to engineer must work in any situation that conforms to such a class of problems.

Requirements and Underspecification Requirements are usually meant to be explicitly identified, quantified, etc, in order to derive from them precise implementable specifications of the solution to be built. But with the type of problems that interest us, it is often the case that requirements cannot be practically and completely specified at a global level because of the complexity of the system and of its context, or because there is no implementable efficient centralised solution. In other words, these problems can suffer from underspecification.

In the robotic example, it is not clear how to specify from the point of view of each bots the global objective of efficiently exploring and securing all victims as the bots cannot communicate across the whole environment, which they don't even know in advance.

From a software architecture point of view, it is known that using (SO)MAS is an architectural response to a certain problems with requirements such as complexity, distribution, dynamicity, scalability, or adaptation [3, 15, 47, 48]. These are other ways of saying that the actual specification of what the system should do changes at runtime and can't be predefined. One way to tackle such underspecification is the use of self-organisation as a way to make the actual solution to the problem be found at runtime through emergence.

In the robotic example, depending on the state of the environment, the victim found and other contextual facts, the bots collective has to devise at runtime the best exploration and securing strategies to follow. If victims are evenly distributed, bots can disperse to secure them all, while if there is big groups of victims, then more bots should dispatch themselves to them, while exploring as much as possible.

Of course, not all requirements are answered through self-organisation and emergence: some of them are answered in a more traditional way while others are answered through emergence. The latter is really the focus here, but choosing which ones should be emergent or not is out of scope of this paper. Nevertheless in the following, we are going to see how to distinguish them when documenting the built system.

Design Constraints It is important to underline the difference between the problem answered by the choice of using self-organisation and emergence, and the constraints this choice implies: most of the works characterising self-organisation and emergence tend to not make this distinction explicit [9, 15, 16, 42].

Indeed, the following can be part of the problem: to have self-adaptation, a distributed deployment context, large-scale system, non-existence of an efficient centralised solution or impossibility of expressing the global behaviour of the system.

Inversely, as said in Section 2.1, the following are mandatory design constraints when embracing self-organisation: the decision must be distributed and decentralised, the global macro-level behaviour and organisation can't be predefined or self-organisation must be a bottom-up process initiated locally by the elements of the system.

Of course, some of these imposed constraints can also be part of the requirements (e.g., distribution of control), but even if they are not (e.g., the system runs on only one computer), they must still be followed when designing a SOMAS.

2.3 Role of the C&C Decomposition Design Activity

In terms of C&C views, designing a SOMAS focuses on two main activities: decomposing in runtime components (the agents) and giving them a self-organising behaviour. For the engineer, these are the two main difficulties to overcome. Often experience and intuition are useful tools to design software systems, but it is accepted that there exist architectural design strategies that can be followed to build good software [4]. Depending on the problem, various strategies are available, and when a strategy is chosen, then it becomes a design constraint for the engineer. We focus on the decomposition activity now.

Following a traditional reductionist approach to software design means to define a global functionality answering the problem requirements, and then, a decomposition in runtime components and their functionalities are chosen *a priori* through a top-down refinement of this predefined functionality. The control is embedded in the way these sub-functionalities are composed: through structural composition (e.g., dataflow components) or by being orchestrated by a central element (e.g., service workflow).

Because of the design constraints highlighted in Section 2.2, such an approach cannot thus be followed if one wants to have adaptation through emergence: the macro-level behaviour is self-designed by a bottom-up autonomous process,

and not by following a top-down human-driven process. This shifts of focus from a traditional design choice as an answer to some requirements to a design choice at the local level of elements to have them find the organisation (i.e., the runtime composition) answering the requirements, is where the paradigm change implied by self-organisation and emergence happens. And this is also where the real difficulty rests: how one can engineer such a micro-level behaviour without thinking about the macro-level behaviour? How to answer the requirements by not directly answering them in our design?

We argue here that the decomposition plays the role of a design bridge between the problem and the emergent behaviour. Indeed, the decomposition in agents implies to choose the local function they play in the system (as it was depicted in the Figure 1): it is the runtime composition of such local functions that makes up the global function, and it is the self-organising behaviour of the agents that leads the system in the adequate composition of the local functions for answering the requirements. Thus, the chosen decomposition constrains the possible macro-level behaviours that can be found at runtime through the self-organising process, and choosing the correct decomposition will determine the success of the engineering of a SOMAS.

2.4 How Should the SOMAS Be Decomposed?

Figure 3 summarises all the relations between the concepts discussed in this section.

It shows the different aspects of the problem answered at design time and runtime by the engineered SOMAS. It shows the two main design activities of decomposition and giving a self-organising behaviour to the agents of the SOMAS. It underlines that the engineer does not tackle at design the requirements emergently answered, but that some constraints have to be followed by the design nevertheless. It also shows that the decomposition and local functions of the agents constrain the possible macro-level behaviours that can emerge at runtime. As the focus of the engineering of SOMAS is on the emergent solution of the problem, we call the local function of the agents their "pre-solved" behaviour because they are solved directly in the humanly designed agent and not emerging at runtime as the Figure shows (a similar differentiation of behaviours is present in the AMAS approach [23] which calls them nominal and cooperative behaviours: the difference is mainly on the adopted problem-oriented point of view).

Finally it highlights the questions that this section raises: how should this decomposition be done? What is a good or a bad decomposition? We give our answer in the next section.

3 Following the Problem Organisation

From our experience in the design of SOMAS and the points discussed in the previous section, we concluded that a strategy to follow when designing SOMAS

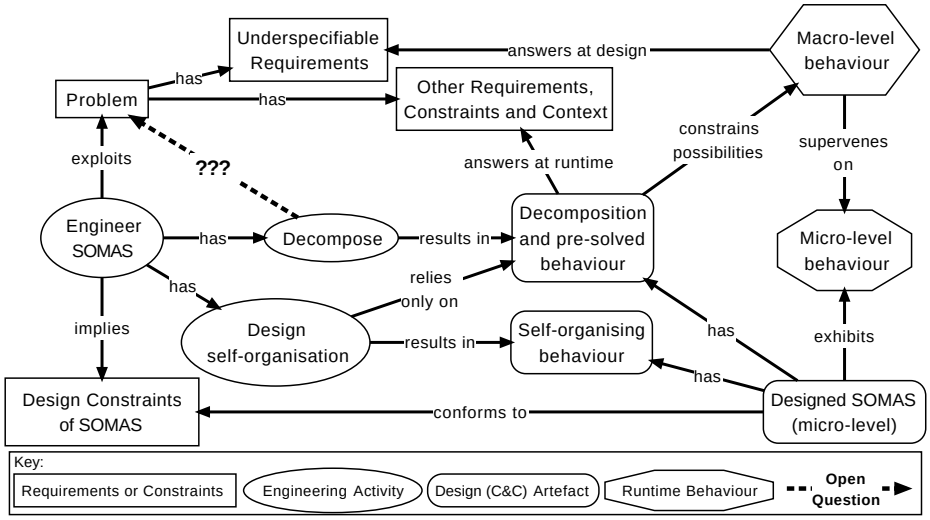


Fig. 3. Relations between Problem, Design Activities and SOMAS.

is to base the software solution on the problem organisation. While we defended previously that decomposition is a design bridge between the problem and the emergent macro-level behaviour, here we explain that following the problem organisation during the decomposition activity is what can make the emergent macro-level behaviour as adapted to the problem as it can be. In the following, we first explain the strategy and present a rational for it. Then we situate it with respect to existing methodological approaches to self-organisation, we underline some open questions, draw lessons for documentation and show architectural advantages for the development.

3.1 The Strategy

It is usual in software engineering to first model the problem space before entering the solution space. However, here we advocate for directly mapping the problem organisation to the solution decomposition, and for only relying on the problem abstractions to design the agents behaviour.

From the Problem Organisation... By problem organisation, we mean the identification of the various elements participating in it and of the role they play with respect to the requirements.

In the robotic example, the elements participating in the problem are the bots, but also the victims and the environment. Furthermore, in relation with the requirements, the elements play the following role in the problem: bots choose a direction to go to, bots communicate with other bots, bots perceive victim, bots perceive the walls, victims are situated, victims need a specific number of bots to be secured, etc.

The Figure 4a shows such a decomposition of the problem in elements. We call it the "organisation" to avoid confusion with the meaning usually associated to a decomposition of the problem in sub-problems as shown in the Figure 4b. The problem organisation can be imposed by the context (that the engineer can't control or modify, as in the robotic example) or must be chosen by the engineer when building the system.

. . . to the Solution Decomposition Based on this modelling of the problem, we then advocate for a direct mapping between elements of the solution (software agents) and the elements of the problem, and for giving them the same capabilities (the local function) as in the problem domain and not more. Their behaviour should be designed locally with respect to the relations elements have in the problem. The decisions (including those of the self-organising behaviour) they take should only rely on the problem domain abstractions and no higher-level global abstractions should be introduced.

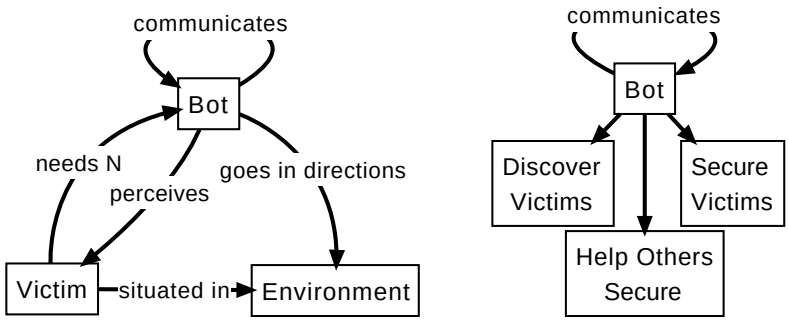
In the robotic example, bots must choose where is the best direction to go at every given moment. For that, they can use what they directly see (victims and explorable areas), and when they don't know what to choose, they should rely on information shared by other bots about the state of the world with respect to the problem: where they are needed for victims or exploration. Hence, bots that see victims or explorable areas advertise about it. This information can be propagated by the bots and they can use it to decide where to go next.

Of course the complexity of the context and of the requirements (e.g., high number of bots, unknown scattering of the victims or limited perception means) are likely to make all these choices difficult. Correctly choosing the best action to take is thus an important questions: we don't pretend to answer it in this paper, but, as said before, we argue that such decisions must rely on the problem domain abstractions. Still, we comment on this question when relating the defended strategy with existing approaches to self-organisation, which propose such guidelines, in the next section.

3.2 Relation to the Design of Self-Organisation

The strategy presented in the previous section can thus be used to design a SOMAS, but, as we highlighted it, is not enough by itself. In particular, a very important point is the problem of taking the correct local decision for the agents.

Some approaches to self-organisation propose guidelines to design the micro-level behaviour of the agents of the system. They propose local criteria to be followed by the agents in order to drive the self-organisation. For example, the work of Gershenson [25] and the AMAS theory [22, 23] are such approaches. In the AMAS approach, the main strategy is that agents must have a cooperative social attitude: the whole approach rests on the theory that if the agents of a system are cooperative with the system environment as well as internally, then the system will behave adequately with respect to the objectives of the agents and of their environment. In the work of Gershenson, the main strategy proposed



(a) In elements ("organisation"). (b) In sub-problems.

Fig. 4. Different decompositions for the robotic problem example.

is to reduce frictions and increase synergy between the agents of the system. In both, most of the actual engineering is about finding which local information to share in order for the agents to act as cooperatively or synergistically as possible. We detail the AMAS approach in the rest of this section (but the same conclusions can be applied to the work of Gershenson) and discuss the coherence of the defended strategy with it.

In the AMAS approach, by identifying local non-cooperative situations agents can face, the engineer designs the agents so that they prevent or correct such situations in order to put the system in a state as cooperative as possible. Usually, a measure called criticality that is shared amongst agents is used to reflect the importance of some state of the problem and to give an agent a way to decide between several choices.

In the robotic example, a bot often has the choice between several directions and do not see any victims. In order to take the most cooperative decision, he needs some information about the state of the system: bots can advertise for example about the direction they chose to go to and some measure (the criticality) of how much more bots are needed in this direction. When a bot propagates this information (because he chose the direction), it will update this criticality in order to reflect his and others participations in the self-organisation process: its choice means that this direction is a bit less critical now. Because bots assume they are all cooperative, they know that a direction chosen by another bot is presently the most important one to go to: choosing the most critical direction amongst all the neighbouring advertisements is enough for a bot to decide where to go next. Every decision taken will then influence how the bot computes this criticality, and inversely.

The way the self-organising process can be designed with this approach heavily relies on the fact that the agents does not contain any pre-defined behaviour in relation to the expected global behaviour, but only concepts manipulated in the definition of the problem itself, which serves to base the local decision on. For example, the criticality measure used in the AMAS approach reflects some aspect

of the problem state in a comparable form: no extra high-level characterisation of what is or not a good global solution is used.

3.3 Rationale

The rationale behind the defended strategy, namely to follow the problem organisation when designing a SOMAS, derives from the points highlighted in Section 2: the global behaviour must emerge, exploiting self-organisation imposes some design constraints and decomposition plays an important role in what can emerge. We discuss here two cases: why the solution shouldn't be based on a decomposition of the problem in sub-problems, and why the solution shouldn't be based on abstractions foreign from the problem organisation. As we are going to see, we do not conclude that these ways of doing are wrong *per se*, but that they have important implications that are usually overlooked.

Sub-Problems A problem decomposition in sub-problems calls for solving each sub-problem separately (if it is not the case, then the decomposition in sub-problems is useless for the design and this is out of the scope of the discussion here). This means that the sub-solutions must then be integrated together, and such integration is embedding the complexity of the problem.

In the robotic example, if the bots have a behaviour to explore and discover victims, and another to go help other bots secure their discovered victims (as in the Figure 4b), it becomes very difficult to handle at the agent level the choice between going in a direction or not: it could be needed to secure a victim, but there may be already other bots going there, so it must gather information about that, and then it could not be needed because other bots are going, but then maybe there is more to explore behind the victim, so it should go anyway, except in the case where there is still enough bots going there for the same reason as it is, and so on...

This puts back the complexity of solving the problem at the agent level instead of making it the result of the collective behaviour: this is the very reason why the paradigm shift proposed by self-organisation and emergence engineering was proposed in the first place. This matter has been discussed many times in the literature (the "complexity bottleneck" [30]) and we don't pretend to bring new arguments for it.

Foreign Abstractions Even if the problem organisation is used to design the SOMAS, the engineer can decide to introduce extra concepts that are foreign to the problem organisation. This means that when facing local decisions, the agents must translate their interpretation of the current state of the problem to the extra abstractions. Going far from the problem implies that we pre-set how situations are interpreted by the agents: it prevents them from interpreting correctly unforeseen situations because the concepts they manipulate can't capture them. In other words, the farther the design is from the problem, the lesser adaptive the system will be, and the lesser adequate behaviours can emerge.

In the robotic example, if the bots are designed so that to explore, they move in the direction of a repulsion vector from other bots (this is a typical algorithm for dispersing bots in an unknown environment), then part of the problem solved is not about exploring while securing anymore, it is about dispersing bots in an environment: for example in a hallway, a stopped bot securing a victim will prevent other bots to go behind him. Inversely, if bots behave as we explained before, when facing a situation where the collective would profit from dispersing, then bots will disperse as a result of going in directions advertised by others where the less bots are going and when facing a situation where there is only one direction to go (e.g., a hallway), bots jut go there because it is the only advertised direction.

3.4 Open Questions

We now look at open questions but don't answer them.

In the light of the discussion, how to make a good modelling of a problem becomes an important question. The strategy and its rationale highlight the impacts of such a decision, but there is many other things to say about it that need more research.

Then, the relation between requirements answered at runtime and the agent's behaviour is not so clear: it seems that it pertains more to strategies for designing the self-organising behaviour than the decomposition and the pre-solved behaviour.

Next, it is obvious that some of the elements identified in the problem can't always be mapped to agents. That means that the other agents are responsible of interpreting the state of such elements and can be many to do so: they must most certainly thus use a common interpretation mechanism (either part of the self-organising or of the pre-solved behaviour) and there may be ways to correctly choose it.

In the robotic example, theoretically, an agent modelling a victim should be responsible of advertising needs for more bots to secure him, but because victims can't be mapped to agents, this responsibility is distributed amongst the bots that locally decide by themselves if they are needed next to a victim as part of their pre-solved behaviour.

Another point is the practical application of the strategy during the design: how much the choices made for the self-organising behaviour impacts the pre-solved behaviour? Does it involve modifying the modelled problem organisation?

3.5 Lessons for Documenting SOMAS

This design strategy and its rationale highlight several points pertaining to documentation and understanding of SOMAS. These "lessons" are useful in our opinion not only when using the proposed strategy but more generally when documenting SOMAS.

First, not following the problem domain and still using self-organisation means that the problem solved is another problem derived from the original.

Second, whichever concept emanating from the problem organisation (including the pre-solved behaviour) and used to design the self-organising behaviour has to be considered as being foreseeable in the problem to solve. Taking these two points into account, during the design and the documentation, is useful to better understand what is actually emergently solved in a SOMAS, and what can be adequately used to build SOMAS (including deciding what should be pre-solved or not).

It is particularly relevant to document the relations between the behaviours and the problem. Some parts of the problem are not solved through self-organisation but directly by the human engineer: they are sub-problems whose solution is implemented by the pre-solved behaviour. The rest of the problem is not directly solved at design, but at runtime by the self-organising behaviour that exploits the state of the problem as well as the pre-solved behaviour capabilities to do so.

In the robotic example, bots move in desired directions while avoiding walls, interpret the needs of the victims they can see, exchange and interpret messages, etc. All of this is pre-solved by the human designer either because there is no need for emergence (for requirements) or because it is imposed as part of the problem (constraints and context).

3.6 Architectural Advantages for Development

The defended strategy promotes some interesting advantages in terms of non-functional requirements. Requirements pertaining to the software itself are well studied in the literature, as seen in Section 2.2, but requirements related to the organisation of the development are not so much. This can be explained by the fact that the existing methodological guidelines in the literature do not directly promote such advantages, while the strategy presented here does. Some of these architectural advantages of the strategy were already identified in the context of MAS [3]: here we improve their rationalisations by linking them to the proposed strategy.

Separation of Concerns First, a design based on the problem tightly reflects the existence of different types of developers of a SOMAS: the developers taking care of the pre-solved behaviour are often expert of the problem domain while those taking care of the self-organising behaviour are often researchers expert in self-organisation. This results in a separation in different implementation modules and thus ease the organisation of the development work.

In the robotic example, roboticians can focus on matters such as vision interpretation, obstacle avoidances, while a self-organisation researcher can exploit these high-level but problem-oriented abstractions.

Furthermore, this is very helpful to well distinguish what are the parts of the behaviour that lead to emergence (the self-organisation behaviour and the aspect of the model it can change) and those that pertain to traditional engineering. The documentation is thus facilitated. Also it is much more difficult to confuse

these two aspects: this avoids making mistakes such as changing the problem to be solved while trying to change the self-organising behaviour.

Maintenance and Incremental Design The way the system is decomposed in various agents that reflects the problem organisation and do not contain sub-solutions to the problem eases the answering of new requirements, either because of a planned incremental design, or unexpected emergent behaviours. Such way of developing is very common from our experience, often because engineering emergence is a bit of an experimental science due to the complexity of the way the system work, and iteration must be done before the system is even in a basic working state.

Handling a new requirement is changing the behaviour of every agents in the same way and not redefining the decomposition in sub-functionality and thus changing the whole architecture. The fact that the system follows directly the problem organisation allows to closely follow the evolutions of the latter: the effort for evolving the software is proportional to the quantity of changes in the problem because of their closeness of structures.

Deployment It is almost obvious that the correspondence between agents and their deployment environment will be very close as the elements are modelled on the problem, which includes the deployment context when there is one.

Rationalisation Rationalising the choices taken during the design is very important when following a proper software architecture approach to development [14]: it is an important enabler for the longevity of a software system in an industrial context for example.

As we presently rationalised the strategy of following the problem for designing the system, this strategy can be used safely when rationalisation is needed, on top of the fact that the advantages presented in this section and in Section 3.5 are useful for explaining and thus rationalising the design choices.

4 Engineering a Swarm of Bots

We now discuss the example with bots exploring and securing victims, which originates from the ASCENS[1] project (see Chapter IV.2 [43]). The proposed SOMAS enables bots to choose where to go and to distribute this decision by exchanging relevant information.

It was fully implemented using the MASON simulator [37] and MAY[2], a tool for supporting the development of MAS using a component-based approach [40]. The sources as well as an executable version are available at the following url: `http://www.irit.fr/~Victor.Noel/unimore-ascens-saso-2014/`.

[1] Autonomic Service Component Ensembles: `http://www.ascens-ist.eu/`.
[2] Make Agents Yourself: `http://www.irit.fr/MAY-en`.

The objective of this section is to illustrate the contributions and not to present the best solution to this problem, but we still show a simple comparison with two known algorithm in order to discuss the validity of the produced design.

4.1 Problem

Bots are situated in a 2D space where they can move in all direction at limited speed. They can identify and localise other bots in their line of sight up to 20m around 360° using a range-and-bearing (R&B) device. They localise victims in their line of sight up to 6m around 360°. They estimate the distance to walls up to 6m in 36 directions with proximity sensors. Finally, the R&B device can advertise data (without size limit): this allows to share information in a local way. All the numbers can be modulated for the sake of experimentation.

The requirements and context are described in Section 2.2. In terms of requirements answered through emergence, while "exploring" isn't something that will emerge, as it is clear that even one bot is acting to explicitly explore, on the other hand it is the collective way to share such exploration that is difficult to achieve and for which no centralised solution exists. The same applies with victims securing: when a bot sees a victim, securing it is not a problem, but what is difficult to achieve is an efficient dispersion of the bots while securing victims quickly as soon as they are found.

4.2 Proposed Design

The swarm exploration example is particularly interesting because, in our opinion, it is not totally straightforward to see how the strategy can be applied: even though it is easy to see that each bot is an agent, the choice of their self-organising behaviour and the type of information it manipulates can lead to very different solutions as highlighted in Section 3.3. It can thus profit from being rationalised, with respect to the arguments given in this paper, and documented as with a proper software architecture approach to development.

Decomposition and Pre-Solved Behaviour This phase is done following the strategy proposed in Section 3.1: see Figure 4a for a simple model of the problem organisation.

The first design choice is about the pre-solved behaviour. Since the problem is about exploring and securing, and since the bot can perceive directions where he can go or not, this behaviour is about choosing a direction to go to, but also about avoiding walls or evaluating state of victims (number of bots needed to secure them). When a victim is seen, the bot will secure it if the number of bots already on it is not enough for the victim's need. When no victim is saw, without any self-organising behaviour, that means choosing randomly one of the available directions.

As we can notice, no prejudgement is done about how the problem (in terms of requirements answered through emergence) is solved yet. Starting from that,

we now construct the self-organising behaviour that enables to tune this local pre-solved behaviour.

Self-Organising Behaviour This phase is done following the AMAS approach while respecting the defended strategy as presented in Section 3.2. Incrementally, we add more and more cooperative behaviour to the agents so that they avoid or correct what is called non-cooperative situations: local situation which are uncooperative with respect to the goal of the agent (i.e., explore and secure). For that they can base their decision on exchanged information, on locally observable information and also on their knowledge of the fact that other bots are cooperative. We simplified the behaviour description to ease the understanding.

Selecting Directions to Consider Bots see walls and free directions around them: since they have some information from bots in that direction, they do not consider by themselves the directions where there is another bot but the advertised direction instead (after translating and estimating the direction for their point of view). Bots see victims, and must advertise about them, but if every bot that sees a victim advertise about it, many bots will propagate duplicate and incorrect information: thus, a bot only advertise about victims without a bot closer to them than it.

Sharing Information Since bots do not see far, they advertise their next movement (a direction) using the R&B device: implicitly, because the bot will act cooperatively, it means that this direction is the best place to go from his point of view. To this direction they associate a measure of criticality as explained in Section 3.2. To a direction with victims is associated a criticality proportional to the number of bots needed in that direction. To an empty area is associated a criticality of one. Hence, in a given instance of the problem, the criticality is upper bounded by the number of bots needed by victims that can be seen in a direction. Then criticality is decreased before being used and propagated in order to take into account the fact that the bot and other bots are now going into the chosen direction. This is not the best in terms of cooperation, but it is enough to illustrate this paper.

Choosing between Directions As said before, they prefer going toward a visible victim than a direction they see or shared by others. The rest of the time, they always choose the most critical direction he knows about, but if there is a tie, they choose the closest to their previous move.

4.3 Observed Global Behaviour

A swarm of bots with these behaviours explores the environment and secure victims. Their behaviour is such that only bots on the borders of the swarm actually consider new directions to explore, while those inside the swarm propagate this information. They start as one set and any explorable direction attracts a

set of bots which separate when facing multiple directions. A victim attracts a small number of bots which stay around it, without attracting too many bots nor preventing them to continue exploration.

4.4 Discussion on the Strategy

In the example, once the problem is known, we kept all reasoning, decision and exchanged information close to the concepts of "choosing a direction" and perceived information to make that decision. We didn't use other abstractions to simplify the reasoning by making it less close to the problem.

The main illustration of that is the criticality that is driving the self-organisation of the system. Thanks to its dynamics, loops in the path of the information exchanged are avoided as fresher information will always takes precedence. Furthermore bots do not "pre-solve" the global problem for the others: the receiver chooses what to do with all the information it receives, and not the sender. Thus, the criticality as it is instantiated in this example is a constantly updated local representation of the most critical aspects of the problem: it does not make any assumption about what is a global solution to the problem and it enables a runtime exploration of the problem space while solving it at the same time.

4.5 Evaluation: Brief Analysis

In order to show that the produced design is good enough to be seriously considered, we compared its performance with two other algorithms. All algorithms (including ours) relies on the same mechanisms for what does not concern self-organisation: wall avoidance, victim's state interpretation and actual securing. We can notice that this corresponds to the pre-solved behaviour of our solution.

Simple Disperse Behaviour Bots consider other visible bots and compute a repulsion vector from them.They do not consider other bots on a victim (the only advertised information) to compute the repulsion vector: it is needed so that stopped bots do not prevent other bots to explore behind them or help them secure victims. They then choose the direction closest to this repulsion vector amongst the directions they can go to. In case of a tie, the closest direction to its previous move is chosen.

Levy Walks As described in [7], bots randomly choose a direction, go in that direction for a random amount of time and when the time is up or they hit a wall or another bot (except if it is to secure a victim), they choose another direction.

Comparisons The performance we compare is time to secure victims (the more secured, the more the algorithm is considered efficient). We ran the algorithms

by modulating various settings: communication range, number of bots, topology of the map. We discuss just some interesting cases.

First, the Levy walks algorithm is bad everywhere (at least with the parameters we used). Then generally, our behaviour is equivalent to the dispersion one when there is a lot of bots (aroung 200) or when the communication range is very low (approx 3 meters). Securing after discovery happens faster with our algorithm (as it is handled by the self-organising process). When the communication range increases too much, the disperse behaviour is less efficient (mainly because bots are too much spaced out and can't see victims in between) while inversely, our behaviour's efficiency increases with the communication range (as they have better information but explore in the same way). When the number of bots decreases, the disperse behaviour efficiency decreases (mainly because they can't cover enough space while staying in contact and thus miss some part of the map) while our behaviour efficiency is more or less unchanged.

What is also interesting is that the design resulting from applying the strategy is completely rationalised and form a coherent whole. With the dispersion algorithm, we had to add some special cases to manage the fact that the repulsion vector was not adequate in some situations, such as when a bot was stopped on a victim in a hallway: this would prevent other bots to go behind the stopped one or helping to secure the victim it is on. Even with these special cases, there is many situations where the disperse behaviour still gives strange results: solving the complexity of the problem has to be done inside the bots behaviours and is not the result of composing self-organising simple bots as with out behaviour. The same comments apply to the Levy walks. Furthermore, the latter was very hard to use because of the parameters that must be tuned by hand for each instance of the problem: we simply didn't succeed doing that, which may explain its bad results.

5 Related Works and Discussion

There exists many research work that can be used to support the engineering of SOMAS, we present and discuss them following various axis.

Applications Some works apply self-organisation to specific problem in order to build SOMAS: many of them can be for example found in the SASO (Self-Adaptive and Self-Organizing systems) community[3]

They don't provide methodological guidelines to help the design of SOMAS, but recurrent practices or self-organising mechanisms can be extracted from them [45]. Nevertheless, it is sometimes implicit in these works that the problem organisation (as we understand it) plays an important role in their functioning: it has even been highlighted in some [34].

[3] See the SASO Conferences at `http://www.saso-conferences.org`.

Reuse and Generic Mechanisms Some works propose self-organising mechanisms that were reused or developed specifically for a problem. Even if it is not always explicitly said in all works on the matter, the idea is that such mechanisms are generic and reusable: they enable to engineer emergence in different contexts than those where the mechanisms were first applied. Many works take inspiration from nature [18] to use well-studied self-organisation mechanisms (famous examples to solve optimisation problem are ant colony optimisation [19], or particle swarm optimisation [35]). Also, some works propose generic (hence reusable) frameworks handling and constraining the self-organising aspects of the system [44], or generic external mechanism to adapt the functioning of the agents at runtime [32].

All these works rely on approaches or mechanisms dependent on a certain class of problems: they have the advantage to be easier to apply and to be reused when possible, but in exchange it is needed to translate the concepts manipulated in the problem to the abstractions of the solution reused. As highlighted in Section 3.5, this means that part of the original problem is lost during that translation: depending on the problem, this can be acceptable or not.

Methods and Modelling Approaches Even though there exist many development methods in the (SO)MAS field [8], very few tackle methodological aspects of engineering emergence itself but focus on other questions not of interest here. We still highlight a type of works that can be mistakenly considered as similar to what is discussed in this paper: ways and models to decompose the problem or the solution. For example, some methods approaches the design with a focus on requirements engineering using goal-oriented notations: an example is the Tropos method [26]. These approaches decompose the problem requirements in goals and sub-goals before attributing them to agents. For example the decomposition shown in Figure 4b is typical in goal-oriented approaches. They do not impose any decomposition in agents, or if they do, it is with a strategy opposite to the one defended here that maps the sub-goals to agents. Another example is role-based decomposition of MAS, used by many methods (the typical example is AgentUML [6]): roles enable to describe a SOMAS and explain its functioning, but the contributions of these works are not about guidelines on what is a good decomposition, which is the main subject of this very paper.

The Ensemble Development Life Cycle proposed within the ASCENS project (see Chapter III.1 [31]), apparently shares a similar endeavour to the above described methodological aspects. However, it has also been explicitly conceived so as to make it possible to easily accommodated problem decomposition approach as the one we have proposed in this chapter.

Design Strategies As discussed in Section 3.2, there exists methodological guidelines, or design strategies, to accompany the design of the self-organising behaviour leading to the emergence of desired properties. For example we cited the work of Gershenson [25] and the AMAS theory [23]. We also redirect the reader to [17] that presents other such strategies. These works help to build the

self-organisation mechanisms themselves as opposite to reuse them (but may of course profit of reuse at other levels of the development), in exchange of a better correspondence between the problem tackled and the built solution. As discussed in Section3.5, this is what can make an emergent behaviour more adapted to a problem.

While this type of works cover well the question of giving a self-organising behaviour to the agents of a SOMAS, very few, if none, works propose clear and rationalised strategies for tackling the decomposition. Nevertheless, the importance of the problem for the decomposition as been sometimes highlighted in these works and as been noted as an important architectural feature of MAS [3]. In the work of Gershenson, an interesting clue is given when self-organisation is defined: it says that "the elements need to divide, but also to integrate, the problem". Similarly, the ADELFE method [10], which supports the applying of the AMAS approach, suggests that the problem domain should help the decomposition in agents. But for all these works, no explanation nor rationalisation are given to support these recommendations.

Experimental Engineering and Simulation As highlighted in Section 3.6, engineering SOMAS is very similar to an experimental science. Some works note that the interleaving of design and simulation can be used to accompany the engineering of emergence in order to iteratively adapt the design with respect to the observed results: they call it "co-development" [2], "using the experimental method" [20] or "disciplined exploration" [41]. Some goes farther with "living design" [24]: designing while the system is running. The discourse of this paper is well coherent with all these approaches, even though they sometimes adopt a predictive understanding of emergence [33]. Nevertheless, our contributions are of a different nature and show that it is possible to exploit the problem organisation to reduce the development effort of SOMAS.

Problem-Orientation In traditional software engineering, exploiting the problem domain is not a novel strategy to approach the development, in particular in the Problem-Oriented Software Engineering (POSE) [28] and the Domain-Driven Design (DDD) [21] fields. DDD is particularly successful in a large scale business context: the focus on problem domain helps to better organise the development and influences the implementation of the system. But complex functionalities are separated from the elements modelling the problem and are often as input on how to model the problem domain, while we advocate for the opposite (when the objective is engineering emergence of course). POSE follows a more academic and formal approach with the objective of helping the human designer to explore the problem space, which is well separated from the solution space.

Thus, an important feature promoted by these works is to exploit the problem space and to decompose it in sub-problems to then better explore the solution space. As we have seen in Section 2, in SOMAS, this same problem space is partially explored by the built system and the roles that the decomposition plays influences the design in different ways. We can also note that some of the

architectural advantages of the strategy presented in Section 3.6 were already identified in these works on problem-orientation, but here we discussed their specificities in the context of self-organisation and emergence. These approaches are thus compatible with the discourse of this paper and a better characterisation of the links with self-organisation would be beneficial to engineers of SOMAS.

6 Conclusion

By revisiting the concepts of self-organisation, emergence and engineering through the lenses of the software architecture and requirements field, we highlighted that the design activity of decomposition in runtime elements plays the role of a design bridge between the problem to solve and the emergent behaviour of the engineered system. We defended the idea that it rationalises the strategy of designing the system by following what we call the problem organisation, and that it is an enabler for the emergent behaviour to be adapted to the requirements. Using the defended strategy gives much advantages, and not only for the system itself but also for the organisation of the project, on top of being fitted for the design constraints imposed by self-organisation and existing approaches.

Self-organisation and emergence are taking more and more important place in today's engineering of software system. The need for clearer and rationalised strategies and methodological guidelines to approach the engineering of emergence is in our opinion an important challenge. There exist many other issues to explore on this subject, such as better way of documenting and distinguishing between requirements answered by emergence from the others, ways to well model the problem tackled with self-organisation or define more precisely how the problem impacts the decentralised decision making.

References

1. Abeywickrama, D.B., Bicocchi, N., Zambonelli, F.: SOTA: Towards a general model for self-adaptive systems. In: WETICE Conference. pp. 48–53. IEEE (2012)
2. Andrews, P., Stepney, S., Winfield, A.: Simulation as an experimental design process for emergent systems. In: EmergeNET4 Workshop: Engineering Emergence (2010)
3. Arcangeli, J.P., Noël, V., Migeon, F.: Software Architectures and Multiagent Systems. In: Oussalah, M. (ed.) Software Architectures, vol. 2, pp. 171–208. Wiley (2014)
4. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, 2nd edn. (2003)
5. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
6. Bauer, B., Müller, J.P., Odell, J.: Agent uml: A formalism for specifying multiagent software systems. International Journal of Software Engineering and Knowledge Engineering 11(3), 207–230 (2001)

7. Beal, J.: Superdiffusive dispersion and mixing of swarms with reactive levy walks. In: International Conference on Self-Adaptive and Self-Organizing Systems. pp. 141–148. IEEE (2013)
8. Bernon, C., Cossentino, M., Pavón, J.: An Overview of Current Trends in European AOSE Research. Informatica 29, 379–390 (2005)
9. Berns, A., Ghosh, S.: Dissecting Self-* Properties. In: International Conference on Self-Adaptive and Self-Organizing Systems. pp. 10–19. IEEE (2009)
10. Bonjean, N., Mefteh Mejri, W., Gleizes, M.P., Maurel, C., Migeon, F.: ADELFE 2.0. In: Cossentino, M., Hilaire, V., Molesini, A., Seidita, V. (eds.) Handbook on Agent-Oriented Design Processes, pp. 19–64. Springer (2013)
11. Bruni, R., Montanari, U., Sammartino, M.: Reconfigurable and Software-Defined Networks of Connectors and Components. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
12. Cabri, G., Puviani, M., Zambonelli, F.: Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In: International Conference on Collaboration Technologies and Systems. pp. 508–515. IEEE (2011)
13. Chalmers, D.: Strong and weak emergence. In: Clayton, P., Davies, P. (eds.) The Re-Emergence of Emergence. Oxford University Press (2006)
14. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2nd edn. (2003)
15. De Wolf, T., Holvoet, T.: Emergence versus self-organisation: different concepts but promising when combined. In: Brueckner, S.A., Di Marzo Serugendo, G., Karageorgos, A., Nagpal, R. (eds.) Engineering Self-Organising Systems, LNCS, vol. 3464, pp. 1–15. Springer (2005)
16. Di Marzo Serugendo, G., Gleizes, M.P., Karageorgos, A.: Self-organisation and emergence in mas: An overview. Informatica 30, 45–54 (2006)
17. Di Marzo Serugendo, G., Gleizes, M.P., Karageorgos, A. (eds.): Self-Organising Software. Natural Computing, Springer (2011)
18. Di Marzo Serugendo, G., Karageorgos, A., Rana, O.F., Zambonelli, F.: Engineering self-organising systems: nature-inspired approaches to software engineering. LNCS (2004)
19. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press (2004)
20. Edmonds, B.: Using the experimental method to produce reliable self-organised systems. In: Brueckner, S.A., Di Marzo Serugendo, G., Karageorgos, A., Nagpal, R. (eds.) Engineering Self-Organising Systems, LNCS, vol. 3464, pp. 84–99. Springer (2005)
21. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley (2004)
22. Georgé, J.P., Edmonds, B., Glize, P.: Making Self-Organising Adaptive Multiagent Systems Work. In: Bergenti, F., Gleizes, M.P., Zombonelli, F. (eds.) Methodologies and Software Engineering for Agent Systems, pp. 319–338. Kluwer (2004)
23. Georgé, J.P., Gleizes, M.P., Camps, V.: Cooperation. In: Di Marzo Serugendo, G., Gleizes, M.P., Karageorgos, A. (eds.) Self-Organising Software, pp. 193–226. Natural Computing, Springer (2011)
24. Georgé, J.P., Picard, G., Gleizes, M.P., Glize, P.: Living Design for Open Computational Systems. In: International Workshop on Theory And Practice of Open Computational Systems at WETICE. pp. 389–394. IEEE (2003)

25. Gershenson, C.: Towards a general methodology for designing self-organizing systems. In: Bogg, J., Geyer, R. (eds.) Complexity, Science and Society. Radcliffe Publishing (2007)
26. Giorgini, P., Kolp, M., Mylopoulos, J., Castro, J.: Tropos: A requirements-driven methodology for agent-oriented software. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 20–45. IGI Global (2005)
27. Goldstein, J.: Emergence as a construct: History and issues. Emergence 1(1), 49–72 (1999)
28. Hall, J., Rapanotti, L., Jackson, M.: Problem-oriented software engineering: Solving the package router control problem. Transactions on Software Engineering 34(2), 226–241 (2008)
29. Heylighen, F.: The science of self-organization and adaptivity. The Encyclopedia of Life Support Systems 5(3), 253–280 (2001)
30. Heylighen, F., Gershenson, C.: The meaning of self-organization in computing. IEEE Intelligent Systems, Section Trends & Controversies 18(4), 72–75 (2003)
31. Hölzl, M., Koch, N., Puviani, M., Wirsing, M., Zambonelli, F.: The EDLC and Best Practises for Collective Adaptive Systems. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
32. Hudson, J., Denzinger, J., Kasinger, H., Bauer, B.: Dependable risk-aware efficiency improvement for self-organizing emergent systems. In: International Conference on Self-Adaptive and Self-Organizing Systems. pp. 11–20. IEEE (2011)
33. Johnson, C.: What are Emergent Properties and How do They Affect the Engineering of Complex Systems? Reliability Engineering and System Safety 91(12), 1475–1481 (2006)
34. Jorquera, T., Georgé, J.P., Gleizes, M.P., Couellan, N., Noël, V., Régis, C.: A Natural Formalism and a Multi-Agent Algorithm for Integrative Multidisciplinary Design Optimization. In: International Workshop on Optimisation in Multi-Agent Systems at AAMAS (2013)
35. Kennedy, J., Eberhart, R.: Particle Swarm Optimization. In: International Conference on Neural Networks. pp. 1942–1948. IEEE (1995)
36. Kim, J.: Making sense of emergence. Philosophical studies 95(1), 3–36 (1999)
37. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: Mason: A multi-agent simulation environment. Simulation: Transactions of the society for Modeling and Simulation International (2005)
38. Mitchell, M.: Complexity: A guided tour. Oxford University Press (2009)
39. Nicola, R.D., Latella, D., Lafuente, A.L., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL Language: Design, Implementation, Verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project, Lecture Notes in Computer Science, vol. 8998. Springer Verlag, Heidelberg (2015)
40. Noël, V.: Component-based Software Architectures and Multi-Agent Systems: Mutual and Complementary Contributions for Supporting Software Development. Ph.D. thesis, Paul Sabatier University (2012)
41. Paunovski, O., Eleftherakis, G., Cowling, T.: Disciplined exploration of emergence using multi-agent simulation framework. Computing and Informatics 28(3), 369–391 (2009)

42. Picard, G., Hübner, J.F., Boissier, O., Gleizes, M.P.: Reorganisation and Self-organisation in Multi-Agent Systems. In: International Workshop on Organizational Modeling. pp. 66–80 (2009)
43. Pinciroli, C., Bonani, M., Mondada, F., Dorigo, M.: Adaptation and Awareness in Robot Ensembles: Scenarios and Algorithms. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project. Springer Verlag, Heidelberg (2015)
44. Pitt, J., Schaumeier, J., Artikis, A.: Axiomatization of socio-economic principles for self-organizing institutions: Concepts, experiments and challenges. Transactions on Autonomous and Adaptive Systems 7(4), 1–39 (2012)
45. Snyder, P., Valetto, G., Fernandez-Marquez, J., Di Marzo Serugendo, G.: Augmenting the repertoire of design patterns for self-organized software by reverse engineering a bio-inspired p2p system. In: International Conference on Self-Adaptive and Self-Organizing Systems. pp. 199–204. IEEE (2012)
46. Weaver, W.: Science and complexity. American scientist 36(4), 536–544 (1948)
47. Weyns, D.: Architecture-Based Design of Multi-Agent Systems. Springer (2010)
48. Weyns, D., Helleboogh, A., Steegmans, E., De Wolf, T., Mertens, K., Boucké, N., Holvoet, T.: Agents are not part of the problem, agents can solve the problem. In: International Workshop on Agent-Oriented Methodologies at OOPSLA. pp. 101–102 (2004)