Advances in complexity engineering

R. Frei*

Intelligent Systems & Networks Group,
Imperial College London,
London SW7 2BT, UK
E-mail: work@reginafrei.ch
*Corresponding author

Giovanna Di Marzo Serugendo

CUI – Université Genève,
Battelle – Bâtiment A, Rte de Drize 7,
CH-1227 Carouge, Switzerland
E-mail: giovanna.dimarzo@unige.ch

Abstract: Complexity science has seen increasing interest in the recent years. Many engineers have discovered that traditional methods come to their limits when coping with complex adaptive systems or autonomous agents. To find alternatives, complexity science can be applied to engineering, resulting in a quickly growing field, referred to as *complexity engineering*. Most current efforts come either from scientists who are interested in bio-inspired methods and working in computer science or mobile robots, or they come from the area of systems engineering. This article is the second part of a set of two articles on this topic; the first one reviewed the definitions of the most important concepts such as emergence and self-organisation from an engineer's perspective, and analysed different types of nature-inspired technology. This article provides a survey of the currently existing approaches to complexity engineering. In the end, challenges ahead are indicated.

Keywords: complex adaptive systems; complexity science; bio-inspired; autonomy; emergence; nature-inspired; engineering; multi-agent systems; self-organisation; self-* properties; robotics.

Reference to this paper should be made as follows: Frei, R. and Di Marzo Serugendo, G. (xxxx) 'Advances in complexity engineering', *Int. J. Bio-Inspired Computation*, Vol. X, No. Y, pp.000–000.

Biographical notes: Regina Frei is currently a Postdoctoral Researcher at the Intelligent Systems and Networks Group, Department of Electrical and Electronic Engineering, Imperial College London, UK. She received her PhD from the Electrical Engineering Department, Faculty of Sciences and Technology, New University of Lisbon, Portugal and her MSc in Micro-Engineering from the Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. Her research interests are self-organising assembly systems, self-* properties and complexity engineering.

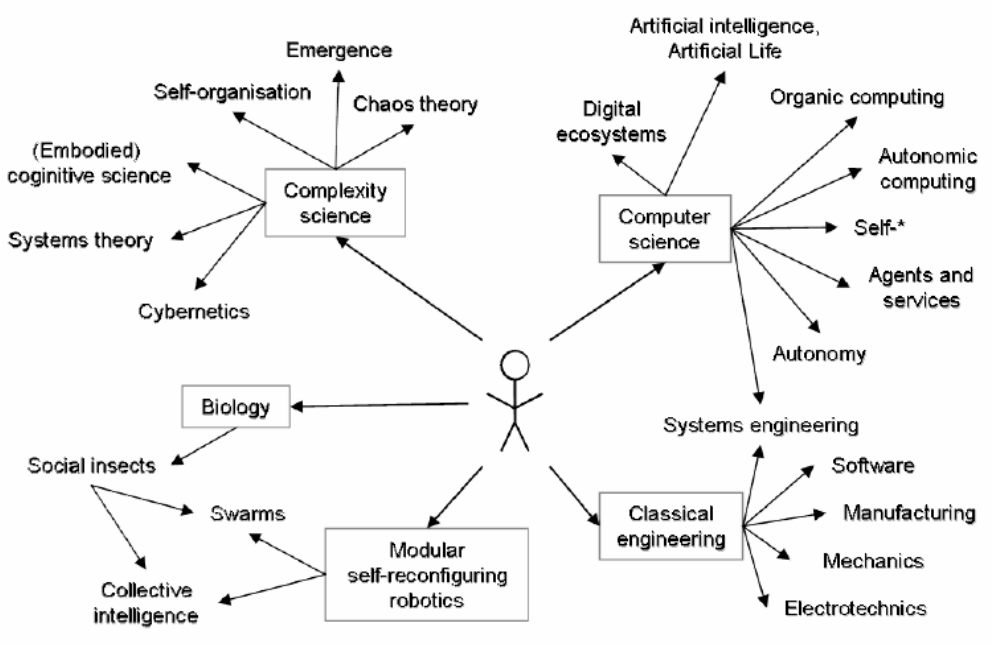
Giovanna Di Marzo Serugendo is a Full Professor at the University of Geneva, Switzerland. From 2005 to 2010, she was a Lecturer at Birkbeck College, London, UK. She received her MSc in Computer Science and in Mathematics from the University of Geneva, Switzerland, and her PhD in Software Engineering from the Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. Her research interests are related to the engineering of self-systems. She co-founded the IEEE International Conference on Self-Adaptive and Self-Organising Systems and is the Editor-in-Chief of the Association for Computing Machinery's Transactions on Autonomous and Adaptive Systems.

1 Introduction

Despite a lot of knowledge about complex systems, the application of this knowledge to the engineering domain remains difficult. Efforts are scattered over many scientific and engineering disciplines such as software engineering, social sciences, economy, physics, chemistry, biotechnology, and others.

Only few of the projects cited in this article have a systematic approach which could be applied to other problems. This lack of general methodologies may have various reasons. Compared to other engineering branches, complexity science is quite recent, and complexity engineering even more so. While researchers observe the typical characteristics of complexity in many different areas, the way of treating them or using them is mostly very individually tailored for the specific system at hand. Furthermore, there is probably a lack of incentives for unifying complexity-related methods, as researchers often rather consider themselves as experts for their area than as complexity engineers.

Figure 1 Complexity-related research areas



There is clearly a need for systematic approaches and generally valid methods. The focus of this article is therefore on how to use the findings of complexity science for engineering, with the most prominent ingredients' being self-organisation and emergence.

Complexity-related research areas: Figure 1 illustrates the situation of the complexity researcher; many different areas are related and relevant for many different types of complex systems in nature and engineering. A multi-disciplinary approach and the ability to communicate with specialists from many different domains are required. How can this overwhelming richness of concepts be managed? Are there useful principles?

First of all, it is necessary to very well understand the characteristics of the system being studied, or the requirements of the systems being engineered (Frei and Barata, 2010). Second, the key concepts for success have to be identified. Most of them cannot be found in traditional engineering disciplines. Third, the concepts and methods taken from non-engineering domains have to be adapted in order to comply with engineering principles.

The inherent multi-disciplinarity requires researchers able of understanding a broad range of concepts, methods and principles. An example of such multi-disciplinarity is natural computing (De Castro, 2006), where natural sciences meet computer science and all kinds of bio-inspired methods are applied to engineering issues. Another example are self-organising assembly systems (Frei, 2010; Frei and Di Marzo Serugendo, 2011), where agile manufacturing comes together with software engineering and complexity science. It is an area that is used for illustrative examples throughout this article.

1.1 Scope and organisation

The topic of this article is engineering, not the sole study of complex systems. We therefore do not discuss *natural* complex systems, but rather consider how to engineer *artificial* complex systems, and how to use the findings of complexity science.

The survey in Section 2 covers work done under the names of *emergence engineering*, *complexity engineering* and other related terms because they mostly address the same type of system and use similar approaches. Typical application areas and concrete cases of complexity engineering are:

- systems engineering: systems of systems in health care, military defence and transportation including pedestrians, bikes, cars, buses, trains and planes
- mobile robotics: swarms for maintenance and safety
- manufacturing automation: agile and evolvable production systems
- software engineering: peer-to-peer, multi-agent systems, safety-critical applications
- communication systems: persuasive computing
- business/finance/economy: prediction and influencing
- nanotechnology and biotechnology: cell engineering, nanorobotics for medical applications.

In Section 3, we reflect on the complexity engineering approach, draw conclusions and give directions for future work.

2 Complexity engineering approaches

This section is organised as follows: concepts and principles are reported in Section 2.1. Section 2.2 details mechanisms and patterns. Modelling and analysis are the subjects of Section 2.3, and Section 2.4 considers design approaches. Section 2.5 details architectures. Methods to develop and implement the designed systems are presented in Section 2.6. Section 2.7 treats validation and verification. Finally, Section 2.8 explains applied approaches.

2.1 Concepts and principles

This section reports a set of concepts and principles which are generally important when creating complex systems. They represent different perspectives which lead to different approaches.

2.1.1 Emergent functionality

Steels (1991) defined emergent functionality (EF) as a function which is not achieved directly by a component or a hierarchical system of components, but indirectly by the interaction of more primitive components among themselves and with the environment. Each component's behaviour has side effects, and the sum of these gives rise to the EF. In order to achieve the desired effect, all the components need to be together and operate simultaneously. Systems with EF are useful when the dependence on the environment is important, and when it is difficult to foresee all possible circumstances in advance. Remark: what Steels called EF is nowadays often referred to as self-organisation; the arguments have mostly stayed the same.

2.1.2 Synthetic ecosystems

Creating systems based on the concept of synthetic ecosystems (Brueckner, 2000; Parunak et al., 1998) or digital ecosystems (Wu and Chang, 2007) are very useful for complexity engineering. Systems are considered 'alife' or 'life-like' and the integration of nature-inspired mechanisms follow almost automatically. The behaviour of species (often insects) and their interactions with each other as well as with the available resources serve as models for with multi-agent systems. The following design principles are suggested (Brueckner, 2000):

1 Things, not functions: avoid functional decomposition take real world units instead.

2 Small agents: prefer many simple agents to a few complicated ones.

3 Diversity, heterogeneity: create agents with differing capabilities and characteristics.

4 Redundancy: the same capabilities should exist more than once, and there should be more than one way to solve a specific problem.

5 Decentralisation: create proactive agents and avoid centralised services.

6 Modularity: it should be possible to compose the system's functionalities stepwise, in layers. [Nevertheless, do not forget the limitations of modularity, discussed in Section 2.1.2 of Frei and Di Marzo Serugendo, (2011)].

7 Parallelism: solve problems in parallel and allow agents to participate in several coalitions at once.

8 Bottom-up control: local interactions lead to a global result, with no entity executing control from the top.

9 Locality: sensor-motor interaction is local, as well as the interactions between the agents.

10 Indirect communication: as far as possible, abstain from direct agent-to-agent communication. Passing messages through a shared environment allows communication to be decoupled in time.

11 Recursion, self-similarity: re-use successful structures and strategies at various levels.

12 Feedback, reinforcement: take into account the result of earlier actions.

13 Randomisation: introduce a random factor in agent decisions to avoid negative synchronism (e.g., all agents heading for the shortest queue at the same instant).

14 Evolutionary change: prefer gradual and evolutionary change to abrupt and revolutionary change.

15 Information sharing: inform other agents. Learn as individuals or as a society.

16 Forgetting: outdated information must disappear automatically.

17 Multiple goals: include maintenance-goals and achievement-goals. Design the system to be able to pursue various goals at once.

An additional design principle, added by the authors of this article, is the use of positive and negative feedback. Their interplay contributes to the system's convergence, oscillations or divergence.

These design rules summarise the most important principles which should always be applied when designing nature-inspired systems. In some cases, there may be reasons for making exceptions, such as having direct communication between the agents. The designer should be aware of the reasons and know that the choice to make an exception may cause difficulties under certain circumstances.

2.1.3 Distributed autonomic computing

De Wolf and Holvoet (2007) suggest that *decentralised autonomic computing* (DAC) can realise autonomic computing in a decentralised way, using emergence. Self-* properties are thus achieved collectively. They propose a taxonomy for self-* properties.

> DAC is achieved when a system is constructed as a group of locally interacting autonomous entities that cooperate to adaptively maintain the desired system-wide behaviour without any external or centralised control (De Wolf and Holvoet, 2007).

DAC is achieved mainly through the implementation of collectively achieved self-* properties, which can be classified according to the following taxonomy criteria (De Wolf and Holvoet, 2007):

- *'Micro versus macro' or 'local versus global'*: self-* properties can be of microscopic (local, concerning a single agent and its immediate vicinity), or macroscopic (global, concerning several agents/the entire system) scope. The way how *locality* is defined is determining for judging if a property is local or global. Additionally, a self-* property can be macroscopic in one system and microscopic in another: it depends on how it is implemented.
- *Ongoing versus one-shot*: most self-* properties are required over an extended time (e.g., maintaining the system protected from malicious intrusion), but there may also be one-shot properties which are triggered from time to time (e.g., self-reconfiguration after major failures).
- *Time/history dependent versus time/history independent*: behaviour which can be objectively measured at any time is time/history independent. Time/history dependent behaviour needs to be seen in relation to the system's evolution over a certain period (e.g., number of packets delivered per hour).
- *Continuous or smooth evolution*: properties which evolve in a smooth way are rather rare. Most of them jump from one state to another.
- *Adaptation-related*: properties which show how well a system adapts to change.
- *Spatial versus non-spatial*: some self-* properties require a spacial structure, while others are not space-related.
- *Resource allocation*: in certain cases the system is required to allocate limited resources to services, or tasks to resources, etc.
- *Group formation*: coalitions or teams may be formed, and also clustering of items or data can be included here.
- *Role-based organisations*: some self-* properties form organisations based on roles and interactions.
- *Self-protection*: some systems need to protect themselves from malicious attacks. This includes defence actions and in certain cases also counter-attacks.

2.2 Mechanisms

According to Bar-Yam (2005), complex systems should be built with strategies modelled after *biological evolution* or *market economics*. Planning mostly does not work in such systems, and design is often done in parallels (*concurrent engineering*). Modularity, abstraction, hierarchy and layering are useful methods, but at some degree of interdependence they become ineffective, as discussed in Section 1.

Other suitable mechanisms include:

- *Trust*: An efficient method for agents to know with whom to collaborate, and whom to avoid, is managing their levels of trust towards their peers. Trust can be established through direct interaction as well as through recommendation from peers who know the agent in question.
- *Gossip*: A difficulty of direct communication is that the receiver of the message must be known in advance. Gossip avoids this, and allows messages to randomly spread across a community.
- *Swarm rules*: Different variants of swarm rules (such as seen in flocks of birds or schools of fish) exist, but they mostly consist of three parts: e.g.:

  1 keep close to your peers
  2 avoid collisions
  3 move forward.

  Such simple, local rules allow any number of agents to act in a coordinated way without requiring any form of centralised control.
- *Stigmergy*: The deposition of markers in the environment is a way of indirect communication often used by social insects, such as ants deposing pheromones. This leads to *collective intelligence* (Bonabeau et al., 1999; Schut, 2010).

Mechanisms generally describe how a process works; patterns (described in 2.2.2) can serve as a more concrete guidance. They define mechanisms in a more systematic way, saying what to do under which conditions.

2.2.1 Friction reduction

Gershenson (2007) proposes that *friction* between interacting agents should be reduced. This will result in a higher *satisfaction* of the system, i.e., better performance. To achieve this, *mediators* can arbitrate among the elements of a system. The goal is to minimise conflict, interferences and frictions as well as to maximise cooperation and synergy. See Table 1 for the possible interactions between two agents *A* and *B*, where the upper part of the table presents strategies for friction reduction, and the lower part strategies for higher satisfaction.

Table 1 Ways to reduce friction/to increase synergy between the elements A and B

Concept	*Explanation*
Tolerance	A shares its resources with B
Courtesy	B searches for alternative resources
Compromise	A combination of tolerance and courtesy
Imposition	Forced courtesy
Eradication	A eliminates B
Apoptosis	B eliminates itself
Cooperation	A and B work together for the benefit of the whole
Individualism	For the benefit of the whole, A can increase its own benefit
Altruism	A can reduce its benefit to the benefit of the whole
Exploitation	Forced altruism

2.2.2 Patterns

Most mechanisms have been expressed as design patterns, which is a way of referencing mechanisms similar to how it is done in software engineering by Gamma et al. (1994).

De Wolf and Holvoet (2007) give some guidance for the design of self-* mechanisms under the form of patterns, including a catalogue of coordination mechanisms which allow the emergence of macroscopic properties. Proposed coordination patterns are:

- *Stigmergy*: indirect communication means communication through the environment. Agents depose, e.g., digital pheromones on their current location, and their peers read the information when passing there. In certain cases, indirect communication is more complicated and less specific than the direct exchange of messages. The main advantage is that communication is decoupled. Agents do not need to respond immediately, or wait for a peer to respond.
- *Gradient-field (also called computational field)*: similar to electric or magnetic fields, computational fields can be sensed by agents who are looking for information or orientation. Notice that gradient-fields can be used to implement other mechanisms, such as stigmergy for task assignment (Weyns et al., 2006) and motion coordination (Mamei et al., 2004).
- *Market-based*: resource allocation is often done by using virtual marketplaces. Agents needing a service make a call for proposals, that offering the service in question answer, and the best offer is selected. This can be done by direct communication, but also works through stigmergy.
- *Tag-based*: tags are observable labels, markings or social cues. They help agents recognise members of a certain group, or agents with a certain characteristic, etc. Tags are especially useful for coordination and group formation.
- *Token-based*: a token is an object which represents the control over a resource or the fulfilling of a role. Token thus exist in limited numbers and are handed from one agent to another when appropriate.

Moreover, Babaoglu et al. (2006) recommend the use of basic biological processes as design patterns in distributed computing:

- *Diffusion*: loose entities tend to naturally spread over a free space. They are transported from an area of high concentration to an area of lower concentration. This mechanism can, i.e., be exploited to let mobile robots distribute themselves over an area.
- *Replication*: cells, viruses or software programmes may create a copy of themselves for various reasons. In computer science, replication refers to the use of redundant resources to improve reliability, fault-tolerance or performance.
- *Chemotaxis*: bacteria and other small living organisms coordinate their movement according to the concentration of chemicals in their environment, i.e., they move towards food sources or away from toxic substances. This concept is related to gradient-fields discussed above.

2.3 Modelling and analysis

The analysis of complex systems is particularly challenging because of the multiple interactions between the components. It is often difficult to detect which components influence each other, and in which ways. There are a few analysis approaches which are specifically made for complex systems, but this does not mean that other approaches may not be suitable as well, if applied with the appropriate care.

Different ways of modelling complex systems take different approaches to solve the problems and have a different focus (Rouse, 2003). The following list is not exhaustive.

- *Hierarchical mappings* refer to the hierarchical decomposition of systems or tasks into simpler sub-units. The focus is on modularisation, which is typically used in the classical engineering approach and referred to as *divide and conquer*. As an example, hierarchical mappings could be used to design a car, but they are not very well suited for complex adaptive systems.
- The use of *state equations or differential equations* is a formal method which considers the states in which a system can be. The focus is on how the system gets from one state to another. This is important for cases where the dynamic systems must be controlled in a stable and optimised way, e.g., motors.
- *Non-linear/discontinuous mechanics* focus on simple behaviours which can have chaotic effects. For instance, fluid turbulences can be modelled by non-linear mathematics.

- *Autonomous agents* are naturally suited to model distributed systems where many entities interact in diverse ways. The focus is on the activities of each agent as well as the agent's interactions with each other and the environment.
- *Ecosystems* are typical examples of complex systems (see discussion in Section 2.1.2). When using ecosystems as a model, engineers often refer to them as being *digital*, *synthetic* or *virtual*. The processes in ecosystems 'take advantage of emergence and deliberately mimic evolution to accomplish and manage the engineering outcomes desired' (Norman and Kuras, 2004).
- *Finite element analysis* is a type of numerical analysis, which is typically used to model complicated geometrical structures. Also flows can be modelled with finite elements, e.g., the behaviour of water in a turbine. The focus is on dynamics.
- Schuh et al. (2006) suggest that collaborative systems be modelled as networks, and that there is a difference between *guided networks*, which are explicitly managed by a focal entity, and *self-organised emergent networks*, which are implicitly managed by the context.

2.3.1 Requirements

Design structure networks (DSN) (Woodard, 2006) are a structured approach to linking requirements with design features. DSN help the designer assess the cost of design changes in complex systems. Woodard furthermore suggests *system design games* and a set of agent-based models (the Palm-Handspring model, the value network model and the platform competition model) to analyse design decisions and their consequences. The method is based on the *theory of design evolution* by Baldwin and Clark (2000), which builds on the theory of CAS by Holland (1992). A detailed explanation of Woodard's work would go beyond the scope of this article.

2.3.2 Multi-scale analysis

Multi-scale analysis relates complexity with structure and function. According to Ashby's *law of requisite variety* (Ashby, 1956), at every scale, the variety of the system must be larger than the variety necessary for the task to fulfil. In a generalised form it suggests that the effectiveness of a system organisation can be evaluated by its variety at each scale of tasks to be performed (Bar-Yam, 2003). The limits of this method are given by the ability of a single agent (human being) to understand the interdependencies between the components.

2.3.3 Equation-free macro-scale analysis

Equation-free macroscopic analysis (De Wolf, 2007) serves both analysis and verification. It is mainly usable for swarms and similar collective phenomena which consist of more than one level or scale. While traditional methods focus on the microscale only, this method is adapted for macroscale behaviour.

The equation-free method needs a good microscopic simulation model from which the macroscopic variables can be measured. The strong points of this method is that it is more feasible than formal proofs, founded by dynamical systems theory (which simulations are not), less computationally intensive than a huge number of begin-to-end simulations, and a mixture between individual-based and aggregate-based simulations. It consists of short bursts of microscopic simulations to extract the info which traditional numerical procedures would obtain from direct evaluation of the macroscopic evolution equation, if this equation was available. It requires time-independent converging macroscopic variables (very difficult to find). The method gives statistically relevant info, not about every run of the system.

2.4 Design

The terms *architecture* and *design* are sometimes confused. The architecture (see Section 2.5) is the structure according to which a system is built, whereas the design refers to the process of creating a system (including its architecture). Section 2.4.1 explains design strategies, whereas, Sections 2.4.2 and 2.4.3 report design abstractions.

2.4.1 Design strategies

Marcus (2006) suggests the following design strategies:

- Top-down, which is control-based, with predefined coordination and interactions.
- Bottom-up, which is collaboration-based and self-organising; collaboration and coordination emerge from the interactions.
- 'Middle out', which is coordination-based. It combines existing components and collaborations but also drives new requirements, collaborations and components. It is a mediation between a set or requirements and a set of services, new available capabilities and new needed capabilities.

2.4.2 Information flows

DeWolf (2007) suggests that *information flows* be established between the various *localities* of the system, which means that the designer focuses on which information needs to be available at which location, at which instant, and where it comes from.

2.4.3 Intelligent networks

Rzevski (2004) recommends to create intelligent networks instead of integrated units. This means that intelligence is not inside a single unit but rather emerges from the interactions within the community. The focus should be on

adaptability rather than on stability. Three steps in running a system are identified:

1 sensory perception: detecting and anticipation changes in the environment

2 cognition: reasoning about perceived changes and deciding about the best action

3 execution: controlling the implementation of cognitive decisions.

2.4.4 BASIC

Schut (2010) published a survey on model design for the simulation of collective intelligence. He suggests several levels of model refinement in the design phase, which include problem assessment, modelling (generic, specific and computer model), simulation, verification and validation. The so-called BASIC recipe for modelling consists of determining the following:

1 action set for all individuals

2 observation set for all individuals

3 action $\rightarrow$ observation methods

4 costs for individuals for methods from 3

5 benefits for individuals for methods from 3

6 observation $\rightarrow$ action methods for all individuals

The basic recipe can then be augmented with suitable steps for the actual requirements, such as internal states, diversity, non-determinism or adaptivity, as illustrated in Schut (2010). For the specific modelling, diverse models – available in literature are suggested for typical applications.

2.4.5 Self-made network

Ulieru and Doursat (2010) introduce an approach for the bottom-up evolution of architectures which are based on a self-grown network of basic cells, similar to what happens in embryogenesis. This means that the coding of the behaviours are indirect; they guide the behaviour of the components (the cells), and the behaviour of the system as a whole emerges from their interactions.

Concretely, the system consists of self-assembling nodes which have pairs of attachment nodes and pairs of gradient values, which keep track of the node's position in a chain. The ports can be occupied or free, and if free they can be enabled or disabled. Chains are the simplest self-assembled structures, but also considerably more elaborate ones may emerge.

All nodes carry the same programme with three routines for updating the gradient values, port management and link creation. The parameters given to these routines determine then the topology of the self-assembled structures. Depending on the application, the nodes (or agents) may be given additional characteristics, and they may be heterogeneous.

2.4.6 Genetic programming

Genetic programming consists of taking instructions from programmes and mixing them based on evolutionary algorithms. A reference model for genetic programming was created by Cramer (1985) and formalised by Koza (1992). Fitness functions indirectly represent the global goal of the system; so one might object that the whole process is not emergent in the proper sense. However, it is not given in the fitness function HOW the task is to be solved (Zapf and Weise, 2007). The functions only help to evaluate the adequateness of the solution. The agents finally equipped with the result of the evolutionary algorithm do not have any info about the objective functions neither about the fitness of their current actions. It remains open how to solve the mentioned co-evolution of different agent types, or how to deal with heterogeneous agents.

Zapf and Weise (2007) propose a solution for what they call *offline emergence engineering*, based on a combination of strategies from genetic programming and agent software engineering. In offline approaches, once a programme is generated, there are no changes any more. Group behaviour emerges before it is put into the real environment: simulation is proposed as a mean to find out if the emerging behaviour is appropriate, and if so, the system is realised. An advantage is that evolution within a simulated environment avoids a potentially long learning phase in the real environment. However, such an approach has obvious weaknesses: no simulation is ever going to be complete, and there are always factors influencing the system in reality which were not completely understood at simulation time, or which simply cannot be represented due to their nature.

In the case of *online emergence engineering* (as opposed to offline emergence engineering), Zapf and Weise (2007) suggest that emergence is planned[1] to occur during execution. Nevertheless, through thorough analysis of the components and their multi-lateral interactions, the range of emergent phenomena can certainly be limited, and engineers can design ways for the system to cope with them. For illustration, consider a mobile robot society based on ant-inspired mechanisms: If the rules and mechanisms are evolved beforehand, simulated to be sure that they work, and only implemented afterwards, this is offline engineering. In case the engineer takes basic rules, implements them, and lets them evolve while already running on the real robots, it is online engineering.

2.4.7 Emergence-based engineering

Deguet et al. (2007) describe concepts to build systems that will produce emergent phenomena. Emergence happens between the design and the observation: so-called *design-to-behaviour emergence*. *Downward causation* applied to code and behaviour means that the code/algorithm is determined by the system's behaviour, not the programmer/designer. In other words, the designer gives the machine a description of the expected behaviour and gets some code in return. The main idea is to implement or generate the systems without knowing 'how it works'.

According to Deguet et al. (2007), this can be done by three approaches (each of which is an issue itself!): by imitating phenomena, by using an incremental design process, or by creating self-adaptive systems (and understanding how the (meta-) system will be able to modify itself).

2.5 Architectures

The following architectures are particularly suitable for complex systems.

2.5.1 MetaSelf architecture

MetaSelf (Di Marzo Serugendo et al., 2008, 2010) is a service-oriented architecture for self-organising and self-adaptive systems, where the services are provided by components or agents. This architecture exploits metadata to support decision-making and adaptation, based on the dynamic enforcement of explicitly expressed policies. Metadata and policies are themselves managed by appropriate services. The components, the metadata and the policies are all decoupled from each other and can be dynamically updated or changed.

MetaSelf applications have been made in the area of *dependability explicit computing* (Paes et al., 2007) and *evolvable assembly systems* (Frei et al., 2008).

2.5.2 The autonomic manager

As software systems become increasingly complex and difficult to manage, *autonomic computing* (Kephart and Chess, 2003) was proposed as a way of handling this. Software should actively manage itself instead of passively being managed by a human administrator. Most self-* properties can be achieved under the responsibility of a single autonomous entity (a manager) which controls a hierarchy of other autonomous entities. The autonomic manager consists of a central loop which handles all upcoming events within the system. The *autonomic manager* follows the *MAPE loop* (IBM, 2005), which stands for monitoring, analysis, planning and execution, supported by a knowledge base.

An alternative to this centralised approach is DAC (see Section 2.1.3), where interacting and fairly autonomous individuals replace the manager.

2.5.3 The three-layer architecture

Kramer and Magee (2007) propose a three-layer architecture to realise self-adaptive and self-managing computing systems, where the components configure their interactions themselves. The lowest layer is the *component control*, which includes sensors, actuators and control loops. The middle layer takes care of *change management*. It is a sequencing layer, to which the lower layer reports state changes. New control behaviours are planned here, and parameters for existing control behaviours are adapted. Finally, the highest layer implements the *goal management*. Time consuming planning is executed at this level, according to the change requests coming from the middle layer and the high level goals specified by the user.

2.5.4 Controller/observer architecture

Organic computing (Wuertz, 2008) is a project[2] which combines software engineering with neuroscience and molecular biology. Within this framework, Schoeler and Mueller-Schloer (2005) developed a controller/observer architecture to 'keep emergent behaviour within predefined limits'. It allows the system to make free decisions within so-called *adaptive islands*, limited by pre-set objectives and constraints.

The basic structure consists of an execution unit which receives an input and generates an output. Above the execution unit, there is an observer/controller unit. The observer receives input from the environment as well as from the execution unit. The controller compares the situation reported by the observer to the goals set by the user and reacts by reconfiguring the execution unit.

2.5.5 Task-based adaptation

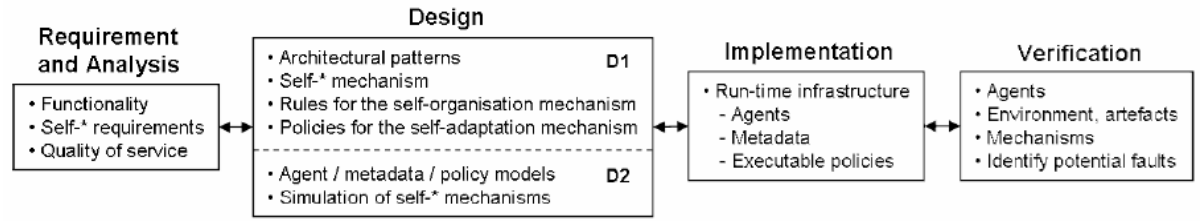
Task-based adaptation (Sousa et al., 2005) is performed by self-adapting computing infrastructures which automate their configuration and reconfiguration. Dynamic task selection can be based on an evolving threshold mechanism and agent stimuli (De Wolf and Holvoet, 2003). External stimuli come from the environment (as it is, not modified by other agents), from interactions with other agents, and in the form of stigmergy (Bonabeau et al., 1999), which is indirect communication, or communication through the environment.

The key ideas in task-based adaptation are:

- explicit representation of user tasks to determine the required service qualities
- decoupling task and preference specification from the low level mechanisms; that is a clean separation between what is needed and how it is carried out
- efficient algorithms to calculate in real-time near-optimal resource allocations and reallocations.

In task-aware systems, the users specify their tasks and goals, and it is the job of the system to automatically map them into the capabilities available in the ubiquitous environment. Computing applications can adapt and reconfigure themselves according to the current tasks to be fulfilled (Sousa et al., 2005; Garlan et al., 2004; Cheng et al., 2006). Such systems automate *human multiple objective trade*-off, considering situation-dependent preferences (knowledge-based decisions).

Figure 2 Final state of the MetaSelf development method



2.6 Development and implementation

This section describes methods for development and implementation of the previously created concepts and architectures.

2.6.1 The customised UP

De Wolf (2007) proposes a design methodology based on the *unified process* (UP) (Jacobson et al., 1999), which is an existing industry-ready software engineering process. The UP was customised to explicitly focus on engineering macroscopic behaviour of self-organised emergent multi-agent systems.

During the *requirement analysis phase* the problem is structured into functional and non-functional requirements, using techniques such as use cases, feature lists and a domain model that reflects the problem domain. Macroscopic requirements (at the global level) are identified. The *design phase* is split into *architectural design* and *detailed design* addressing microscopic issues. *Information flow* (a design abstraction) traverses the system and forms feedback loops. *Locality* is "that limited part of the system for which the information located there is directly accessible to the entity" (De Wolf, 2007). Activity diagrams are used to determine when a certain behaviour starts and what its inputs are. Information flows are enabled by decentralised coordination mechanisms, defined by provided design patterns. During the *implementation phase*, the design is realised by using a specific language. When implementing, the programmer focuses on the microscopic level of the system (agent behaviour). In the *testing and verification phase*, agent-based simulations are combined with numerical analysis algorithms for dynamical systems verification at macro-level.

The CUP approach has been applied to autonomous guided vehicles and document clustering (De Wolf, 2007).

2.6.2 Policies and metadata

A way to guide a system in its development without hard-coding its behaviour is the use of policies, as suggested by Kephart and Walsh (2004), and Kephart and Das (2007) in the context of autonomic computing (Kephart and Chess, 2003). Policies can express actions, goals and utility functions. Depending on their type, they lead one or several agents to directly execute an action (i.e., if the gripper blocks, try to re-initialise it), to maintain their behaviour as to reach a certain goal (e.g., always keep the speed below 3 m/s), or to follow a more complicated guideline and choose appropriate actions (such as: reduce the effort of reconfiguration).

Policies always work in conjunction with corresponding metadata, which is data that is not directly processed in operation. Metadata can describe the performance of an axis, the interfaces of a gripper, the preferential partners of a mobile robot or the current availability of a GPS module.

2.6.3 MetaSelf design method

The MetaSelf development method (Di Marzo Serugendo and Frei, 2009; Di Marzo Serugendo et al., 2010), which consists of four phases, is illustrated in Figure 2.

The *requirement* and *analysis phase* identifies the functionality of the system along with self-* requirements specifying where and when self-organisation or self-management is needed or desired. The required quality of service is determined.

The *design phase* consists of two sub-phases. In the first part, *D1*, the designer chooses architectural patterns (e.g., autonomic manager or observer/controller architecture) and self-* mechanisms, governing the components' interactions and behaviour [e.g., trust, gossip, or stigmergy, that is indirect coordination through changes in the environment (Bonabeau et al., 1999)]. Rules for self-organisation and policies for self-adaptation are defined. In the second part, *D2*, the individual autonomous components (services, agents, etc.) are designed. The necessary metadata and policies are selected and described. The self-* mechanisms are simulated and possibly adapted/improved.

The *implementation phase* produces the run-time infrastructure including agents or services, metadata and executable policies.

In the *verification phase*, the designer makes sure that agents, the environment, artefacts and mechanisms work as desired. Potential faults and their consequences are identified, similar to the way *failure modes and effects analysis* (FMEA) (McDermott et al., 2008) works, and measures to avoid the identified faults are taken accordingly.

2.6.4 Evolutionary engineering

In Bar-Yam's (2003, 2005) *evolutionary engineering (EE)/enlightened evolutionary engineering* (E^3), the advances a system makes are often unanticipated and not fully understood, but the system does *learning by doing*. Evolutionary processes are based on incremental iterative

change and cyclical feedback. EE includes methods which involve rapid parallel exploration and a context designed to promote change through competition between design/implementation groups, with field testing of multiple variants. Examples of evolutionary methods in software engineering are: *spiral development*, *extreme programming* and the *open source* movement. The functioning products which are in use at a certain moment in time are considered as the evolving population which will be replaced by new generations of products. If the function of a system needs to change, the system can adapt because there are many possible variants of subsystems that can be generated. The focus of E^3 is on creating environment and process rather than a product, and it continually builds on what already exists. Operational systems include multiple versions of functional components, and E^3 uses multiple parallel development processes. More effective components are gradually introduced.

Bar-Yam proposes the following methods:

1 Analysing the environment and temporarily modifying it to influence the complex system's self-directed development. (Complex systems cannot be completely isolated from their environments).

2 Tailoring developmental methods to specific scales and regimes (i.e., phases in the life-cycle of a complex system, such as development and operation).

3 Identifying or defining a targeted outcome space at multiple scales and in multiple regimes. (Outcome spaces are close to specifying 'requirements' or 'desired capabilities' for complex systems).

4 Establishing rewards and penalties, including the explicit formulation of satisfying criteria. (Not to confound with direction and guidance, which directly concern agent behaviour; rewards and penalties refer to agent generated outcomes).

5 Judging actual results and allocate prices. This is associated with the criteria of rewards but also involves the explicit consideration of other outcomes.

6 Formulating and applying developmental stimulants.

7 Characterising continuously, i.e., capturing and publishing information about the way things are at every moment in a complex system. Among others, this helps agents take decisions and allows tracking the evolution of the system.

8 Formulating and enforcing safety regulations (policing).

Related to EE, and maybe better-known, is *evolutionary computation* (De Jong, 2006). It belongs to the field of *artificial intelligence*; it is mostly concerned with optimisation tasks and uses the mechanisms of evolutionary reproduction and inheritance. Evolutionary computation is not to be confused with *genetic programming* (Section 2.4.6).

2.6.5 The AMAS theory and ADELFE

Engineering systems which generate emergent functionalities is the goal of Capera et al. (2004) and Gleizes et al. (2007). Their adaptive multi-agent system (AMAS) theory claims that for any functionally adequate system, there exists at least one cooperative internal medium system that fulfils an equivalent function in the same environment. ADELFE (Bernon et al., 2005) is an engineering methodology for AMASs, based on the AMAS theory. ADELFE is limited to cooperative systems and does not provide support for the achievement of specific goals.

The main ADELFE strategy is to maintain cooperation, or in other words, to avoid so-called *non-cooperative situations* (NCS). Agents try to anticipate these NCS, and act accordingly. This means that designers have to describe their own specific NCS set and plan the respective actions for each kind of agent. Notice that it is certainly not always possible to preview all the NCS which can occur, and designing corrective actions for them is not easy, neither.

A cooperative agent in the AMAS theory has the following characteristics: it is autonomous; it is unaware of the global function of the system (this emerges from the agent level towards the multi-agent level); it can detect NCSs and acts to return in a cooperative state; it is not altruistic but benevolent (it seeks to achieve its goal while being cooperative).

2.6.6 A general methodology

The general methodology by Gershenson (2007) provides guidelines for system development. Particular attention is given to the vocabulary used to describe self-organising systems. It is composed of five iterative steps or phases: representation, modelling, simulation, application and evaluation.

In the *representation phase*, according to given constraints and requirements, the designer chooses an appropriate vocabulary, the abstractions level, granularity, variables, and interactions that have to be taken into account during system development. Then, the system is divided into elements by identifying semi-independent modules, with internal goals and dynamics, and with interactions with the environment. The representation of the system should consider different level of abstractions.

In the *modelling phase*, a control mechanism is defined, which should be internal and distributed to ensure the proper interaction between the elements of the system, and produce the desired performance. However, the mechanism cannot have strict control over a self-organising system; it can only steer it. To develop such a control mechanism, the designer should find aspects or constraints that will prevent the negative interferences between elements (*reduce friction*) and promote positive interferences (*promote synergy*). The control mechanism needs to be adaptive, able to cope with changes within and outside the system (i.e., be *robust*) and active in the search of solutions. It will not necessarily maximise the satisfaction of the agents, but rather of the system. It can also act on a system by bounding

or promoting randomness, noise, and variability. A mediator should synchronise the agents to minimise waiting times.

In the *simulation phase*, the developed model(s) are implemented and different scenarios and mediator strategies are tested. Simulation development proceeds in stages: from abstract to particular. The models are progressively simulated, and based on the results, the models are refined and simulated again. The *application phase* is used to develop and test model(s) in a real system. Finally, in the *evaluation phase*, the performances of the new system are measured and compared with the performances of previous ones.

This methodology was applied to traffic lights, self-organising bureaucracies and self-organising artefacts (Gershenson, 2007)

2.6.7 Agents and artefacts meta-model

Gardelli et al. (2008) use architectural pattern based on the *agents and artefacts* (A&A) *metamodel* which features agents as proactive goal-driven entities, and artefacts as encapsulated services to be exploited by agents. The environment plays an important role in this approach. It consists of artefacts and environmental agents, which are incorporated self-organisation mechanisms. These environmental agents are responsible for sustaining feedback loops between the agents and the environment.

This approach consists of three iterative design stages: modelling, simulation and tuning. In the modelling phase, the agents' behaviour is designed, and architectural structures are sketched. Afterwards, simulation is used to verify the suitability of the agents and the architecture. In the tuning phase, parameters are adapted in order to optimise the system's performance.

2.7 Validation and verification

After creating solutions at micro level, the system verification mainly aims at giving guarantees that the resulting macroscopic behaviour meets the requirements (De Wolf, 2007). This is almost never straight-forward. For instance, software code cannot prove to be correct, or to have been exposed to all relevant environmental scenarios. It is thus appropriate to talk about *acceptable behaviour* (Zapf and Weise, 2007), or to give more detailed indications about the verified scenarios.

Most of the approaches which have been proposed for modelling in Section 2.3) can also be used for validation and verification purposes, in particular those in Sections 2.3.2 and 2.3.3. Sometimes, macroscopic behaviour can only be verified by begin-to-end simulations; efforts to formalise emergence are typically limited to rather simple application scenarios (De Wolf, 2007). But as simulations are always abstractions of reality, they alone are often not enough to prove that a complex engineered system will comply with the requirements. Especially self-organisation and emergence challenge researchers. Different subsystems depend on and interact with each other in many often very complex, dynamic and unpredictable ways.

Not all verification methods are equally useful for any case. Most often a combination of different methods will do best. De Wolf and Holvoet (2007) propose the methods represented in Table 2, together with their typical applications.

Table 2 Verification methods

Method	*Application*
Unit-based and integration testing	Most useful for one-shot microscopic properties
Formal proof	Microscopic; not usable for interaction models
Statistical experimental verification	Long-term ongoing properties Expensive due to large number of experiments
Equation-based macroscopic verification	Adaptation-related; only if the macroscopic property in question can be modelled as a variable in a (partial) differential equation
Equation-free macroscopic verification	Long-term ongoing properties with smooth and continuous behaviour, adaptation-related; time-dependent variable reflecting the property in question has to be found (see Section 2.3.3)
Time series analysis based on chaos theory	Adaptation-related long-term behaviour, measuring complexity, i.e.

2.8 Applied approaches

A view of complex systems engineering from the perspective of *integrated circuit design evolution* was given by Bramlett (2002). It seems that, different from other perspectives, for CPU design, component coupling is important, and the systems are considered as closed and highly optimised. The design process can be seen as a series of phase transitions in convergence towards design requirements, which is an emergent property. Abstractions at different levels and granularities are used to define convergence phases, rates and transitions. Often the design process itself is far more complex than the artefact it produces. The author also states that there is a need for open architectures for cross-disciplinary engineering, taking the human as part of the system.

Rzevski (2004) presents complexity engineering at the example of an *intelligent variable geometry compressor* and a family of *space exploration robots*); however, some theoretical background about the used strategies may be missing. Remarkably, Rzevski's strategy for *self-repair* is isolating defective parts and thus making them harmless. Such an approach certainly makes sense in practice, but it does not correspond to the usual interpretation of the term *reparation* as it does not repair the defect neither consider the consequences of an isolation on the rest of the system.

3 Discussion, conclusions and directions

After reviewing numerous existing concepts and methods in complexity engineering, we now analyse the general situation.

3.1 Discussion

The methods and approaches cited in this article are mostly from the area of computer science. This is due to the nature of complexity engineering: the systems in question usually need some kind of intelligence and a corresponding control system, which leads us typically to computer science. Purely mechanical systems are rarely complex and adaptive or self-organised.

The claim that decentralised control should be avoided has been quite prominently uttered in the last few years. But is it always favourable to build a system with purely decentralised control? Decentralised systems also have weaknesses. They are often not optimal, they take longer to solve problems, and may use more resources to do so. On the positive side, distributed systems are robuster and they can better cope with disturbances.

Due to their nature, the validation of complex systems with emergence and self-* properties is difficult. Formal approaches at agent level do not automatically cover global phenomena. Simulations are another way to verify system behaviour, but there are the obvious limitations of time requirements and non-completeness to this approach (Gleizes et al., 2007). Formal modelling techniques can capture important features of the design choices and enable designers to reason about them in a useful way (Woodard, 2006). We may have to accept that we will never be able to completely control or predict the behaviour of a complex system; we should rather cope with this by adapting our actions to the new situations (Gershenson, 2007). This indicated that deterministic models or predictions are not necessary; having realistic default expectations with the possibility to correct errors or exceptions after they have occurred, works quite well in practice.

3.2 Conclusions

The application of complexity engineering methods should always be accompanied by a reflection on the reasons why these methods have been chosen. Are they useful for the actual application? Or might other methods be more suitable? The engineering method should always be selected with care.

A fundamental challenge of complexity engineering is that it touches many different domains; it is therefore difficult to decide about generally applicable methods. For instance, network models and statistics may be helpful when creating wireless communication systems but not at all for building manufacturing systems. This article tries to structure the existing methods and thus make it easier for engineers to choose a method which is suitable for their applications.

This article ends with directions for further research which we consider important for the development of complexity engineering.

3.3 Further research directions

Complexity engineering has still not been established as a proper engineering domain. Research remains scattered and focused on specific examples, which is the reason why most methodologies are not generally applicable. We would like to encourage other researchers to make efforts in complexity engineering, and to coordinate their research with peers. A general framework for complexity engineering should be created, linking existing and new methods with each other, giving receipts for how to approach which type of problem. Complexity engineering requires particular attention concerning the following issues (Buchli and Santini, 2005): theory, universal principles, implementation substrates, designing, programming and controlling methodologies as well as collecting and sharing of experience.

Although academia increasingly discovers their interest in complexity engineering, industry is reluctant. It is difficult to persuade industrials to give away total control. Complexity is mostly perceived as disturbing, annoying or overwhelming. Researchers should therefore not only develop methodologies for complexity engineering, but at the same time also try to persuade industry of the benefits which using complexity can offer.

Industry requires dependable methods. Self-organised emergent MAS will only be acceptable in an industrial application if one can give guarantees about the macroscopic behaviour (De Wolf, 2007). This can be shown experimentally or proven formally. Both formal prove and experimental evidence has advantages and disadvantages. On one hand, experiments often provide statistical evidence that the desired results will often appear under certain circumstances. But it can also mean that the adverse conditions which lead to failure have not been encountered yet. Formal proof, on the other hand, always uses abstractions, and making the right abstractions is difficult. Formal proofs are useful for understanding certain aspects of a system, but they can never express the complete reality. Additionally, they depend on the language chosen to describe the system. Every language has a certain expressivity. This expressivity may be suitable for certain aspects of a system, but limit the model in capturing others.

Methods to provide sufficient evidence of dependability should be developed especially for complexity engineering methods, given that they are often different from traditional engineering methods due to the use of self-* properties and emergence.

Acknowledgements

This work was started while Regina Frei received his PhD Grant from the Portuguese Foundation for Science and Technology. She currently receives a post-doc grant from the Swiss National Science Foundation.

References

Ashby, W. (1956) *An Introduction to Cybernetics*, Chapman & Hall, London.

Babaoglu, O., Canright, G., Deutsch, A., Caro, G., Ducatelle, F., Gambardella, L., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A. and Urnes, T. (2006) 'Design patterns from biology for distributed computing', *ACM Transactions on Autonomous and Adaptive Systems*, Vol. 1, No. 1, pp.26–66.

Baldwin, C. and Clark, K. (2000) *Design Rules, Vol. 1: The Power of Modularity*, MIT Press, Cambridge, MA, USA.

Bar-Yam, Y. (2003) 'When systems engineering fails – toward complex systems engineering', in *IEEE Int. Conf. on Systems, Man & Cybernetics (SMC)*, Vol. 2, pp.2021–2028, Washington DC, USA.

Bar-Yam, Y. (2005) 'About engineering complex systems: multiscale analysis and evolutionary engineering', in Brueckner, S., Di Marzo Serugendo, G., Karageorgos, A. and Nagpal, R. (Eds.): *Engineering Self-organising Systems: Methodologies and Applications, ESOA 2004, LNCS*, Vol. 3464, pp.16–31, Springer Berlin.

Bernon, C., Camps, V., Gleizes, M-P. and Picard, G. (2005) 'Engineering adaptive multi-agent systems: the Adelfe methodology', in Henderson-Sellers, B. and Giorgini, P. (Eds.): *Agent-Oriented Methodologies*, pp.172–202, Idea Group Pub., Hershey, PA, USA.

Bonabeau, E., Dorigo, M. and Théraulaz, G. (1999) *Swarm Intelligence*, Oxford University Press, New York, USA.

Bramlett, B. (2002) 'Engineering emergence', Tech. rep., MIT Media Lab, Cambridge, MA, USA.

Brueckner, S. (2000) 'Return from the ant – synthetic ecosystems for manufacturing control', PhD thesis, Institute of Computer Science, Humboldt-University, Berlin, Germany.

Buchli, J. and Santini, C. (2005) 'Complexity engineering, harnessing emergent phenomena as opportunities for engineering', Tech. rep., Santa Fé Institute Complex Systems Summer School, NM, USA.

Capera, D., Picard, G. and Gleizes, M-P. (2004) 'Applying ADELFE methodology to a mechanism design problem', in *Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, Vol. 3, pp.1508–1509, New York, USA.

Cheng, S-W., Garlan, D. and Schmerl, B. (2006) 'Architecture-based self-adaptation in the presence of multiple objectives', *in ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp.2–8, Shanghai, China.

Cramer, N. (1985) 'A representation for the adaptive generation of simple sequential programs', in *Int. Conf. on Genetic Algorithms and their Applications*, pp.183–187, Mahwah, NJ, USA.

De Castro, L. (2006) *Fundamentals of Natural Computing*, Chapman & Hall/CRC Computer and Information Sciences, New York, USA.

De Jong, K. (2006) *Evolutionary Computation: A Unified Approach*, MIT Press, Cambridge, MA, USA.

De Wolf, T. (2007) 'Analysing and engineering self-organising emergent applications', PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium.

De Wolf, T. and Holvoet, T. (2003) 'Adaptive behaviour based on evolving thresholds with feedback', in *AISB, 3rd Conf. on Adaptive Agents and Multi-Agent Systems (AAMAS)*, pp.91–96, Melbourne, Australia.

De Wolf, T. and Holvoet, T. (2007) 'A taxonomy for self-* properties in decentralised autonomic computing', in Parashar, M. and Hariri, S. (Eds.): *Autonomic Computing: Concepts, Infrastructure, and Applications*, CRC Press, Taylor and Francis Group, pp.101–120.

Deguet, J., Magnin, L. and Demazeau, Y. (2007) 'Emergence and software development based on a survey of emergence definitions', *Studies in Computational Intelligence*, Vol. 56, pp.13–21.

Di Marzo Serugendo, G. and Frei, R. (2009) 'Experience report in developing and applying a method for self-organisation to agile manufacturing', Tech. rep., BBKCS-09-06, School of Computer Science and Information Systems, Birbeck College, London, UK.

Di Marzo Serugendo, G., Fitzgerald, J. and Romanovsky, A. (2010) 'Metaself – an architecture and development method for dependable self-* systems', in *Symp. on Applied Computing (SAC)*, pp.457–461, Sion, Switzerland.

Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A. and Guelfi, N. (2008) 'Metaself – a framework for designing and controlling self-adaptive and self-organising systems', Tech. rep., BBKCS-08-08, School of Computer Science and Information Systems, Birkbeck College, London, UK.

Frei, R. (2010) 'Self-organisation in evolvable assembly systems', PhD thesis, Department of Electrical Engineering, Faculty of Science and Technology, Universidade Nova de Lisboa, Portugal.

Frei, R. and Barata, J. (2010) 'Distributed systems – from natural to engineered: three phases of inspiration by nature', *Int. J. of Bio-inspired Computation*, Vol. 2, Nos. 3/4, pp.258–270.

Frei, R. and Di Marzo Serugendo, G. (2011) 'Self-organising assembly systems', Accepted for publication in *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*.

Frei, R., Di Marzo Serugendo, G. and Barata, J. (2008) 'Designing self-organization for evolvable assembly systems', in *IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO)*, pp.97–106, Venice, Italy.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional Computing Series, Boston, USA.

Gardelli, L., Viroli, M., Casadei, M. and Omicini, A. (2008) 'Designing self-organising environments with agents and artifacts: a simulation-driven approach', *Int. Journal of Agent-Oriented Software Engineering*, Vol. 2, No. 2, pp.171–195.

Garlan, D., Poladian, V., Schmerl, B. and Sousa, J. (2004) 'Task-based self-adaptation', in *Workshop on Self-healing Systems, 1st ACM SIGSOFT Workshop on Self-managed Systems*, pp.54–57, Newport Beach, CA, USA.

Gershenson, C. (2007) 'Design and control of self-organizing systems', PhD thesis, Faculty of Science and Center Leo Apostel for Interdisciplinary Studies, Vrije Universiteit, Brussels, Belgium.

Gleizes, M-P., Camps, V., George, J-P. and Capera, D. (2007) 'Engineering systems which generate emergent functionalities', in *Engineering Environment-Mediated Multiagent Systems – Satellite Conf. held at The European Conf. on Complex Systems (EEMMAS 2007)*, Dresden, Germany.

Holland, J. (1992) *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, MIT Press, Cambridge, MA, USA.

IBM (2005) 'An architectural blueprint for autonomic computing', Tech. Rep. June.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*, Addison Wesley, Reading, MA, USA.

Kephart, J. and Chess, D. (2003) 'The vision of autonomic computing', *IEEE Computer*, Vol. 36, No. 1, pp.41–50.

Kephart, J. and Das, R. (2007) 'Achieving self-management via utility functions', *IEEE Internet Computing*, Vol. 11, No. 1, pp.40–48.

Kephart, J. and Walsh, W. (2004) 'An artificial intelligence perspective on autonomic computing policies', in *Proc. 5th IEEE Int. Workshop on Policies for Distributed Systems and Networks (POLICY)*, pp.3–12, New York, USA.

Koza, J. (1992) *Genetic Programming, on the Programming of Computers by Means of Natural Selection*, A Bradford Book, The MIT Press, Cambridge, MA, USA.

Kramer, J. and Magee, J. (2007) 'Self-managed systems: an architectural challenge', in *Future of Software Engineering (FOSE)*, pp.259–268, IEEE Computer Society, Washington, DC, USA.

Mamei, M., Zambonelli, F. and Leonardi, L. (2004) 'Cofields: a physically inspired approach to motion coordination', *IEEE Pervasive Computing*, April–June, Vol. 3, No. 2, pp.52–61.

Marcus, R. (2006) 'Complex systems engineering for the global information grid', available at http://cs.calstatela.edu/wiki/images/a/a4/Marcus.ppt.

McDermott, R., Mikulak, R. and Beauregard, M. (2008) *The Basics of FMEA*, CRC Press, Taylor & Francis Group, New York, USA.

Norman, D. and Kuras, M. (2004) 'Engineering complex systems', available at http://www.mitre.org.

Paes, R., Lucena, C. and Carvalho, G. (2007) 'Using interaction laws to implement dependability explicit computing in open multi-agent systems', in *Brasilian Symposium on Software Engineering (SBES)*, pp.59–75, Joao Pessoa, Brazil.

Parunak, H.V.D., Sauter, J. and Clark, S. (1998) 'Toward the specification and design of industrial synthetic ecosystems', in *4th Int. Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages (ATAL)*, pp.45–59, Springer-Verlag, London, UK.

Rouse, W. (2003) 'Engineering complex systems: implications for research in systems engineering', *IEEE Transactions on Systems, Man and Cybernetics – Part C: Applications and Reviews*, Vol. 33, No. 2, pp.154–156.

Rzevski, G. (2004) 'Designing complex engineering systems', in *Volga Conf. on Complex Adaptive Systems*, Keynote paper, Samara, Russia.

Schoeler, T. and Mueller-Schloer, C. (2005) 'An observer/controller architecture for adaptive reconfigurable stacks', in *Int. Conf. on Architecture of Computing Systems (ARCS)*, pp.139–153, Innsbruck, Austria.

Schuh, G., Sauer, A. and Dring, S. (2006) 'Modeling collaborations as complex systems', in *4th Int. Industrial Simulation Conf. (ISC)*, pp.168–174, Palermo, Italy.

Schut, M. (2010) 'On model design for simulation of collective intelligence', *Information Sciences*, Vol. 180, pp.132–155.

Sousa, J., Poladian, V., Garlan, D., Schmerl, B. and Shaw, M. (2005) 'Task-based adaptation for ubiquitous computing', *IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews*, Vol. 36, No. 3, pp.328–340.

Steels, L. (1991) 'Towards a theory of emergent functionality', in Meyer, J-A. and Wilson, S. (Eds.): *From Animals to Animats: 1st Int. Conf. on Simulation of Adaptive Behaviour*, pp.451–461, Paris, France.

Ulieru, M. and Doursat, R. (2010) 'Emergent engineering: a radical paradigm shift', *J. of Autonomous and Adaptive Communications Systems*, to appear.

Weyns, D., Boucke, N. and Holvoet, T. (2006) 'Gradient field-based task assignment in an AGV transportation system', in *Proc. of 5ht Int. Conf. on Autonomous Agents*, pp.842–849, ACM, New York, NY, USA.

Woodard, C. (2006) 'Architectural strategy and design evolution in complex engineered systems', PhD thesis, Business Studies Department, Harvard Univ., Cambridge, MA, USA.

Wu, C. and Chang, E. (2007) 'Exploring a digital ecosystem conceptual model and its simulation prototype', in *IEEE Int. Symp. on Industrial Electronics (ISIE)*, pp.2933–2938, Vigo, Spain.

Wuertz, R. (Ed.) (2008) *Organic Computing. Understanding Complex Systems*, Springer, Berlin Heidelberg.

Zapf, M. and Weise, T. (2007) 'Offline emergence engineering for agent societies', in *Proc. of the Fifth European Workshop on Multi-Agent Systems EUMAS'07*, Hammamet, Tunesia.

Notes

1 This might be considered as a contradiction in itself: emergence can hardly be planned.

2 Available at http://www.organic-computing.org.