

**Emergence as a Paradigm to Engineer  
Distributed Autonomic Software**

*Tom De Wolf  
Tom Holvoet  
Yolande Berbers*

*Report CW 380, March 2004*

**Katholieke Universiteit Leuven**  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Emergence as a Paradigm to Engineer Distributed Autonomic Software

*Tom De Wolf*  
*Tom Holvoet*  
*Yolande Berbers*

*Report CW 380, March 2004*

Department of Computer Science, K.U.Leuven

## **Abstract**

Today's software systems are becoming more and more complex, i.e. distributed, situated, open, and dynamic. In addition, there is a trend towards software that deals with its complexity autonomously - the term autonomic computing has been coined to reflect this system behaviour. Making distributed systems autonomic means constructing these systems as a group of interacting autonomous entities that are expected to cooperate. The ultimate challenge in engineering distributed autonomic systems is to find approaches to achieve globally coherent behaviour. This paper describes emergence, a biologically inspired paradigm, as a promising approach for addressing this challenge. Emergence is an innovative way of analysing and reasoning upon globally coherent behaviour that "emerges" from the local interactions between the individual entities. Employing emergence in a distributed system results in fundamental implications on software engineering. This paper outlines the characteristics of distributed autonomic computing systems, the motivation for considering emergence in this context, and the implications this has on software engineering.

**Keywords :** emergence, autonomic computing, software engineering.

**AMS(MOS) Classification :** Primary : D.2.0, Secondary : C.2.4, C.5.0, A.1.

# Emergence as a Paradigm to Engineer Distributed Autonomic Software

## A biologically inspired paradigm

Tom De Wolf, Tom Holvoet  
and Yolande Berbers  
Department of Computer Science  
KULeuven  
Celestijnenlaan 200A  
3000 Leuven, Belgium

{Tom.DeWolf, Tom.Holvoet}@cs.kuleuven.ac.be

### ABSTRACT

Today's software systems are becoming more and more complex, i.e. distributed, situated, open, and dynamic. In addition, there is a trend towards software that deals with its complexity autonomously - the term *autonomic computing* has been coined to reflect this system behaviour. Making distributed systems autonomic means constructing these systems as a group of interacting autonomous entities that are expected to cooperate. The ultimate challenge in engineering distributed autonomic systems is to find approaches to achieve globally coherent behaviour. This paper describes emergence, a biologically inspired paradigm, as a promising approach for addressing this challenge. Emergence is an innovative way of analysing and reasoning upon globally coherent behaviour that "emerges" from the local interactions between the individual entities. Employing emergence in a distributed system results in fundamental implications on software engineering. This paper outlines the characteristics of distributed autonomic computing systems, the motivation for considering emergence in this context, and the implications this has on software engineering.

### 1. INTRODUCTION

A trend in today's computing systems is that they are becoming more and more complex, i.e. distributed, situated, open, dynamic and these systems have a high degree of locality. Examples are telecommunication networks, flexible manufacturing networks, urban traffic networks, economic systems such as supply chains, Automated Guided Vehicle material transportation, and simulation of ecological systems.

On top of that evolution, there is a trend towards more

and more autonomic computing systems. The paradigm of *autonomic computing* aims to design and build computing systems capable of adjusting to varying circumstances by taking the initiative to decide and to do things themselves. It is a requirement that autonomic computing systems anticipate needs autonomously. The real complexity is hidden from the user. In the autonomic computing perspective this is described by the goal to achieve the self-X principles (see [18]).

The increased complexity implies that systems need to handle more and more complexity on their own. Making increasingly distributed systems autonomous implies using individual autonomous entities inside the system. The increased distribution implies that only decentralised local control can be used for scalability reasons. The ultimate challenge is to find approaches to achieve global coherent behaviour in distributed autonomic computing systems. This paper describes *emergence* as a promising approach that matches this profile.

The use of emergence as a software paradigm for distributed autonomic systems has a number of fundamental implications on the way software is engineered. These implications result in a lot of future work to cope with the challenges of emergent software engineering.

In the first section of this paper the shift in computing systems is described to give an idea of the systems we must be able to engineer in the future, the properties that characterise those systems, and the challenges imposed by them. Then, in the next section we define and describe the characteristics of emergence to show that the biologically inspired paradigm of emergence is a promising view on computing that enables us to exploit and cope with the challenges of tomorrow's software. In the third section, the implications on software engineering are described. In other words, what is important when building software for distributed autonomic computing based on emergence. And finally, future work to achieve the vision of this paper is described and after that we summarise this paper.

## 2. SHIFT IN COMPUTING SYSTEMS

In this section we describe a two-way shift in today's computing systems. First, we see that systems are more and more inherently distributed with a lot of interaction. Secondly, the impact of autonomic computing is considered. Next to these major shifts, some other characteristics are considered that are important to describe the kind of systems we have to deal with today and in the future (partially inspired by [45]). And finally, a number of challenges towards engineering such computing systems are formulated.

### 2.1 Distribution and Interaction

Systems become more and more distributed. This includes physical distribution, due to the invasion of networks in every system, and logical distribution, because there is more and more interaction between applications on a single system and between entities inside a single application.

Due to the inherent distribution of computing systems, *interaction* becomes an important and essential part of computing systems. What we see is that there is a shift in problem solving power from computations inside each entity towards problem solving power from more and more interactions between entities.

In [42] interaction is shown to be more powerful than algorithms for computer problem solving, overturning the prevalent view that all computing is expressible as algorithms. Some things can't be done without interaction such as network routing, on-line supply chain management and agile manufacturing. This radical notion that interactive systems are more powerful problem-solving engines than algorithms was the basis for new paradigms for computing technology built around the concept of interaction (e.g. object-orientation, agent-orientation).

Because interactions can be more powerful than algorithms or computations inside an entity, interaction is more and more a dominant feature with respect to individual computations and actions of the subparts of the system. For example, in interaction-centred manufacturing, each machine is responsible for only a small part of the whole production process. Having one machine that manufactures the whole product is not flexible enough. The interaction between the machines is much more important and the work done by one machine or entity in the system is relatively simple.

Extrapolating this evolution towards more and more distribution and interaction into the future results in a shift from systems with complex individual entities and few interactions towards systems consisting out of relatively simple individual entities and a huge number of complex interactions.

### 2.2 Autonomy

Current requirements for computing systems often impose the need for the system to do things autonomously. A very simple example is that you want your computer to check your e-mail automatically every 10 minutes or so. But in industrial applications this is also the case: network routing has to be done autonomously, on-line supply chain management is not useful if the system does not take initiatives autonomously, etc .

The fact that systems will become so complex that a human user cannot operate the system manually anymore also implies the need for more autonomy. This complexity must be hidden from the users so that they can concentrate on what they want to accomplish rather than figure out how to manipulate the computing systems to get the appropriate behaviour. The complex system has to be operated autonomously.

This trend towards even complete autonomy is called *autonomic computing* in a vision statement by IBM [18]. The paradigm of autonomic computing aims to design and build computing systems capable of adjusting to varying circumstances by taking the initiative to decide and to do things themselves. It is a requirement that autonomic computing systems anticipate needs autonomously. In the autonomic computing perspective, this is described by the goal to achieve the self-X principles (see [18]): self-healing, self-management, self-protection, self-configuration, self-organisation, etc .

In a distributed computing system, to have a completely autonomous system, the individual entities must be autonomous. Due to the fact that an individual entity is autonomous it can decide on its own when to (inter)act and how to (inter)act. The latter results in uncertainty that must be taken into account by the system. This will be discussed later.

### 2.3 Other Characteristics

#### 2.3.1 *Situatedness and Openness*

Computing systems become more and more situated, i.e. there is an explicit notion of the environment, the context in which the system and its entities exist and execute (in this paper we use 'context' and 'environment' as synonyms). Environmental characteristics affect their execution and often there is explicit interaction with that environment. For example, indirect coordination and communication between entities in the system can be done through the environment, i.e. a middle-ware platform. Such an environment has its own dynamics, independent of the intrinsic dynamics of the individual entities and/or the computing system.

Being situated in an environment, perceiving it, and being affected by it intrinsically implies a fuzzy and open boundary between the system and its environment. The system is no longer isolated, but becomes a permeable subsystem, whose boundaries permit reciprocal side-effects that are often complex.

Such a fuzzy and open boundary, i.e. openness, can result in structural dynamics, i.e. dynamic changes in the system structure (systems or entities in a system can move between environments, interaction channels change constantly and thus interacting with the same entity is unlikely,...).

#### 2.3.2 *Locality in control*

The 'flow of control' concept has always been one of the key concepts of computer science and software engineering, at all levels, from the hardware level up to the high-level design of applications. When software systems live and interact in an open world and consist out of multiple autonomous entities, the concept of flow of control is still essential but it becomes

less fundamental with respect to the global behaviour of the system. That global behaviour is no longer the execution of one global flow of control.

Independent software entities have their own autonomous flow of control, and their mutual interactions do not imply any join of these flows. We evolve from systems where the global behaviour is achieved by following a global flow of control towards systems where there are only multiple local flows of control. Those local flows do not explicitly incorporate the global behaviour of the system. The global behaviour implicitly arises from the interactions between those local flows of control.

### 2.3.3 Locality in interaction

When we have a physical environment (e.g. robot application), locality of interaction is automatically enforced by physical laws. In a logical environment (e.g. a computer network), if we want to minimise the complexity of managing and building software entities, i.e. achieving simple entities with high cohesion and low coupling, we must favour modelling the system in local terms and limiting the effect of a single entity on the environment. With respect to interaction this implies using only local interactions between software entities.

Locality in interaction is also a strong requirement when the number of entities in a system increases, or as the dimension of the distribution scale increases. In the latter case, it is even necessary for scalability that one entity does not have a high coupling with too many other entities in the system.

### 2.3.4 Highly Dynamic

As pointed out a number of times in the previous characteristics, systems are becoming more and more dynamic in a number of ways such as dynamics from the environment, structural dynamics, and huge interaction dynamics.

From a software engineering point of view the rapidly changing requirements also cause a need for high dynamics. For example, in manufacturing systems, the competitive pressures are reducing product life times and thus forcing systems to change frequently. Also, machine failures and upgrades force the system to adapt to these changes. In such a situation, the system needs to be very flexible and dynamic.

## 2.4 Challenges

In the previous sections, we listed a number of characteristics of distributed autonomic computing systems. From these characteristics some challenges, to cope with when engineering such systems, can be extracted:

- The *high degree of interaction* due to increasing distribution implies a shift towards systems consisting out of (simple) entities that make a huge number of complex interactions. The problem-solving power of the system needs to come more and more from the interactions and less from the computations inside the individual entities. Exploiting this is the challenge here.
- There is a *higher degree of uncertainty and dynamics*. Tracking and controlling concurrent and autonomously

initiated interactions is much more difficult than in object-oriented and component-based applications. The reason for this is that autonomous entities can decide on what kind of interactions are executed and when those interactions occur. Thus, this information cannot be determined and used when engineering such systems.

There is also uncertainty due to the increased situatedness and openness. Building context-aware systems implies that we must be able to cope with the uncertainty and unpredictability about the environmental dynamics.

And finally all the dynamics, i.e. rapidly changing context, structure, and requirements, require a self-adapting system. The system needs to cope autonomously with these changes and the uncertainty related with these changes.

- Another challenge is a *higher degree of locality*. Locality in control implies that controlling the global behaviour by following a global flow of control becomes impossible. We can only use local flows of control and the interactions between them. In other words, we can explicitly control the actions of the parts, not the whole. The whole must be controlled with local means.

In addition, locality in interaction implies that we can only use local information to achieve local control because global information about the system is not available. This of course increases the problem.

- All the previous challenges and characteristics make it difficult to understand and control the global behaviour of the system. Thus, the biggest challenge to cope with is the *need for a global coherent behaviour* of the computing system.

The need for a global coherent behaviour is nothing new. This need is also present in traditional object-oriented systems. In that case the presence of a single flow of control makes it possible to follow that global flow of control and thus to control the global behaviour of that system explicitly. In distributed autonomic computing systems, the entities are more than objects, i.e. they become autonomous. In that case, the flow of control stays local in each entity and thus a global flow of control is not available. This makes it more difficult to engineer a global coherent behaviour.

There already exists research where the systems evolve towards systems where the global flow of control is less pronounced and/or disappears. For example, distributed objects emphasise distribution. But there is often still a global flow of control that visits each distributed entity. Another example are active objects which have a weak form of autonomy. Objects are often considered active when they allow asynchronous behaviour. Other objects can send messages to the active object so that method returns immediately while the work that must be done is performed autonomously by the active object. In other words, there are objects that are considered active only for this reason and such objects still do nothing if they do not receive any messages. Real autonomy would require that the invocation of activity is also internal to the entity, i.e. no external messages are needed [26]. Thus, in distributed

autonomic computing, the entities have a stronger autonomy and combined with distribution a global flow of control disappears. This makes it difficult to engineer a global coherent behaviour.

### 3. EMERGENCE AS A PARADIGM

As stated earlier, the problem is to get a global coherent behaviour of your software that is constructed as a group of autonomous interacting entities. Logically this global behaviour has to *emerge* from the (inter)actions of the individual entities. The local, individual behaviour gives rise to a global behaviour. When this global behaviour acts as a coherent whole and no single individual entity has explicit knowledge of the global behaviour then this phenomenon is called '*Emergence*'. The inspiration for it originated from biology (see [3]) where colonies of ants interact and reach a global coherent behaviour. In this section a definition of emergence is given and then we argue why the concept of emergence is so important with respect to the challenges of distributed autonomic computing.

#### 3.1 Definition

In literature, many definitions of emergence are described, each with their own 'important characteristics' [16, 4, 10, 14, 15, 25, 26, 32]. Without stating that the following definition is the only correct definition of emergence, we attempted to cover the most important and recurring characteristics in our working definition:

Emergence is a dynamic, nonlinear process that leads to emergents (properties, behaviour, structure, patterns, ...) at the macro-level or global level of a system from the (inter)action of the parts at the micro-level or local level. Such emergents are novel, i.e. they cannot be readily understood by taking the system apart and looking at the parts (=reductionism). They can however be studied by looking at each of the parts in the context of the system as a whole.

The 'level' mentioned in the definition refers to certain points of view towards the system. The macro-level considers the system as a whole and the micro-level considers the system from the point of view of the individual entities that make up the system.

A simple example from biology to clarify this is an ant colony. At the micro-level each ant observes its local environment and performs local actions accordingly. The macro-level is looking at the system as a whole. For example, path formation with pheromones is an emergent that is situated at the macro-level of the system. When we look at a single ant, it can perceive and follow the pheromones but it has no clue at all about the existence of paths starting from the food sources and ending at the nest.

Some important characteristics of emergence are the following:

- **Micro-macro effect** [16, 4, 10, 14, 15, 25, 26, 32]: this is the most important characteristic and is

mentioned explicitly in most definitions in literature. This effect refers to the emergence of a macro-level behaviour from the (inter)actions between entities at the micro-level.

- **Radical novelty** [4, 14, 10, 2]: this refers to the non-reductionism to the individual entities. The global behaviour is novel w.r.t. the individual behaviours at the micro-level. At the micro-level there is no explicit representation of the global behaviour.
- **Coherence** [10, 26, 25]: emergents appear as integrated wholes that tend to maintain some sense of identity over time (i.e. a persistent pattern). Coherence spans and correlates the separate lower level components into a higher level unity.
- **Dynamical** [10, 15]: emergents arise as a complex system evolves in time. In a rapidly changing context the emergent behaviour is very dynamic. Changes influence the global behaviour and in order to maintain the global behaviour there must be a constant dynamic that handles these changes at the micro-level.
- **Nonlinearity** [10, 15]: emergence requires the "small cause, large effect" principle and has an intense focus on nonlinear interactivity. Nonlinearity makes it possible to get those secondary effects at the macro-level that we call emergents.
- **Decentralised Control** [25, 15]: you control the action of the parts, not the whole.
- **Interacting parts** [25]: the parts must be interacting - parallelism is not enough. Without interactions, interesting macro-level behaviours will never arise.
- **Robustness and Flexibility** [25, 15]: emergents are relatively insensitive to perturbations or errors, and have a strong capacity to restore themselves. Increasing damage will decrease performance, but degradation will be 'graceful'. For example, the failure of a single entity will not cause a complete failure of the entire system.

All these characteristics are relevant in distributed autonomic computing systems. The next section elaborates on this and argues why emergence is so important in that context.

#### 3.2 Why Emergence?

The properties of emergence and the challenges stated earlier, that result from a shift in computing system characteristics, match remarkably well. Emergence is a promising paradigm to exploit when engineering such systems. In this section we elaborate on this by considering each challenge separately and by describing why emergence has common properties with distributed autonomic computing systems.

##### 3.2.1 Higher Degree of Interaction

The fact that there is a shift towards getting more problem-solving power from the interactions instead of internal computing supports the paradigm of emergence. This is because interesting macro-level behaviour cannot arise without interacting parts. Also, the property of nonlinearity is focussed

on nonlinear interactivity. Emergence is based on interaction as an enabling mechanism. This means that a high degree of interaction in distributed autonomous computing systems can become an advantage instead of a disadvantage. It can be exploited to achieve emergent behaviour.

### 3.2.2 Higher Degree of Uncertainty and Dynamics

A higher degree of uncertainty due to high dynamics in distributed autonomous computing means that we need to anticipate this uncertainty. Systems need to take this uncertainty factor into account. Emergent-based systems have the advantage of locality in this matter. There only needs to be anticipation for this uncertainty at the level of the individual entities. An autonomous entity cannot assume that every activity it performs will succeed and/or have the normal intended effect.

Emergence is also robust and flexible with respect to uncertainty due to unanticipated events. For example, a failure of a single individual entity does not imply a complete failure of the emergent system behaviour.

Thinking and engineering rapidly changing systems in terms of emergence implies an intrinsic dynamics in such systems. Those dynamics are essential to have a coherent global behaviour. There must be a constant dynamic at the micro-level of the system to maintain the emergent behaviour in the presence of changes and perturbations.

The different kinds of dynamics in a system, i.e. dynamics from the context or environment, interactions, and actions done by the autonomous entities, can interfere with each other. This way, an action, interaction, or dynamic can have an uncertain effect. For example, when an autonomous entity moves to a position in a spatial system the effect will be that this entity is located at that position. However, when two entities move to the same position, this is not possible and one of the autonomous entities will not succeed in its move. That entity must be able to cope with this uncertainty and failure. In the context of emergent systems, individual entities need to maintain a global behaviour and they need to constantly inspect the situation and act accordingly. Thus, each effect that results from uncertain interference between different kinds of dynamics will be taken into account when the individual entities decide on their next (inter)action. In other words, context-awareness is essential to cope with uncertainty.

### 3.2.3 Higher Degree of Locality

Emergence is based on the property of decentralised control. Decentralised control means that we use only local influences or controls to reach a wanted effect on the global behaviour of the system. In systems where we only have local flows of control, there is no other possibility than to initiate a control from a locally situated autonomous entity. The effect of this decentralised control or influence can then, for example, propagate throughout the whole system without violating the need for local control and interaction.

Another property of emergence is the radical novelty of the emergent with respect to the individual entities. This implies that not a single entity in the system can have knowledge about what the global behaviour should be. The knowl-

edge about the global wanted behaviour is not explicitly represented in any part of the system. The power of emergence is that this knowledge is implicitly present in the local relatively simple entities and their interactions. When considering large distributed autonomous computing systems, keeping that knowledge completely distributed, local, and implicit is also a requirement to engineer scalable and flexible systems. For example, a global plan of the behaviour in an entity makes that entity a bottleneck and single point of failure for the whole system.

### 3.2.4 Need for a Global Coherent Behaviour

As mentioned before, the biggest challenge is to engineer the systems with a global coherent behaviour. And this is the biggest advantage of emergence: emergents appear as integrated and coherent wholes over time. The cause for this is the micro-macro effect where the macro-level is a coherent unit and the micro-level is a very dynamic and rapidly adapting level.

How a specific global coherent behaviour can be engineered based on emergence is still an open issue. This section only stated that there is a remarkable match between the properties of the paradigm of emergence and the properties of distributed autonomous computing systems. In the next sections of this paper, the implications on software engineering of an approach based on emergence are outlined.

## 4. IMPLICATIONS ON SOFTWARE ENGINEERING

In a report of the Software Engineering Institute (SEI) [8] engineering is defined as the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind. Software engineering is defined as that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems. In short, we can say that this refers to how we can build software.

In the previous sections we described the characteristics of distributed autonomous computing systems, i.e. tomorrow's systems, and the biologically inspired paradigm called Emergence that matches remarkably well on the challenges that such computing systems have. In this section we describe the implications on software engineering, that result from the characteristics and challenges of emergence and distributed autonomous computing. In other words, what is important when applying the principles of computer science and mathematics to achieve a coherent global behaviour for distributed autonomous computing software based on the paradigm of emergence?

To support the implications below, we often refer to literature in the multi-agent community. In general, a lot of research in the multi-agent community is dealing with software systems that have the characteristics of distributed autonomous computing and emergence. For example, agents are autonomous entities, they interact a lot, there is an approach in the agents community called 'situated multi-agent systems' where the importance of situatedness is emphasised, when multi-agent systems are engineered without central

control they also rely on emergence for a global coherent behaviour, etc ... . This justifies why later in this paper references from that community are used to support statements given.

## 4.1 Interaction-Oriented

First of all we need to put more emphasis on the use of interaction and we need to exploit interaction patterns and mechanisms. In systems where the global coherent behaviour has to result from the (inter)actions of individual entities, the internal computations of those entities will become less important. We have to get our problem-solving power by exploiting interaction as a key part of software engineering.

For example, in integrated supply chains, each company is responsible for only a small part of the whole production process towards the customer. Having one company that manufactures the whole product is not flexible enough. The interaction and integration between the individual companies is much more important to get a flexible and efficient supply chain.

Another application domain where the interaction is a dominant feature is in so called sensor-webs. A sensor web is an independent network of wireless, communicating sensor nodes, deployed to monitor and explore a range of environments [21, 6, 7]. This adaptable instrument can be tailored to the conditions it is sent to observe. Each individual sensor does nothing more than sensing its environment. To reach the goal of the sensor web, i.e. aggregation of information, the sensors need to interact. They must propagate the sensed information to the nodes where the information is aggregated and inspected. By using different interaction mechanisms that complement each other we can exploit those interactions to aggregate the information efficiently. For example, the nodes that aggregate and inspect the information can propagate 'interest-packets' through the web. The flow of such packets puts a gradient on the sensor web, i.e. the direction in which the packet was received is 'up-the-gradient'. The sensors that receive these packets then know in which direction the information can be sent towards the gathering point. This way, sensors send their information to other sensors that are closer to the gathering point.

The following quote from a call for papers on Agent-Oriented Software Engineering supports the importance of putting more emphasis on the interaction instead of the computation [9]:

Over the past three decades, software engineers have derived a progressively better understanding of the characteristics of complexity in software. It is now widely recognised that interaction is probably the most important single characteristic of complex software. Software architectures that contain many dynamically interacting components, each with their own thread of control, and engaging in complex coordination protocols, are typically orders of magnitude more complex to correctly and efficiently engineer than those that simply compute a function of some input through a single thread of control.

Unfortunately, it turns out that many (if not most) real-world applications have precisely these characteristics. As a consequence, a major research topic in computer science over at least the past two decades has been the development of tools and techniques to model, understand, and implement systems in which interaction is the norm. Indeed, many researchers now believe that in future, computation itself will be understood as chiefly as a process of interaction.

To summarise, an interaction-oriented approach to software engineering is needed. Therefore, the interaction should be present in every aspect of software engineering such as the methodology, the modelling approach, and the technology used. To give an idea, multi-agent systems can be a supporting technology, as stated in [17]. Here the central software abstraction is an agent, i.e. an autonomous and interacting software component. There are also modelling techniques to describe interactions between agents, for example by using and extending UML [27], that allows to incorporate the notion of interaction as an essential part within the software development life cycle. But this is not enough. More is needed to support the full development life cycle.

## 4.2 Autonomy-Oriented

The main consequence of autonomic computing on software engineering is of course the presence of more autonomy in software. Software has to take the initiative and handle complexity autonomously. For example, in an on-line supply chain management [34, 36] application for a network of connected companies it is not feasible for a human to gather enough information in time in order to control and steer that supply chain efficiently. Therefore, the supply chain must be supported by an autonomous management system which anticipates the needs and constraints of each company in that chain.

The lack of support in current software engineering methodologies for autonomous entities inside a system is a problem. These methodologies are more user-centric, and use-case based. However, next to regular information input use cases, autonomic software has only one use case to steer the system and that is starting the system. After that it has to meet its requirements autonomously without further user-interaction to control and steer the system. A possible idea is not to put the user in the centre of the software engineering process, i.e. use case driven, but to put the autonomous entity in the centre, i.e. autonomy driven. For example, instead of use cases we could have autonomy cases that describe the interaction between the autonomous entity and the context in which it is situated (i.e. other autonomous entities, the environment, objects in that environment, etc ...). Such an autonomy case is then started when the participating autonomous entity takes the initiative to start it.

Another concern that has its influence on the way we engineer software is the fact that numerous autonomous entities can run in parallel. We have to engineer our software with that in mind. Perhaps, there is a need for concurrent programming with, for example, a technology like threads. One step further, systems can use scheduling techniques to

ensure fairness with respect to the number of actions an autonomous entity can do compared to other autonomous entities. Another promising approach is to specify domain-specific time models that make it possible to ensure causality relationships between actions that are specific for a certain application domain [13]. Such time models can be specified by the designer of the application. An underlying platform can then take this time model as an input and ensure the timing issues specified in that model.

### 4.3 Decentralised Control

Because of more distribution and autonomy we shift from software where we have one global flow of control that can be engineered to get the wanted behaviour towards software consisting out of multiple local flows of control that interact with each other. The main goal in engineering such systems is to incorporate control mechanisms so that the wanted behaviour is achieved. A large majority of software implemented today falls into the category of centralised control. This approach requires global knowledge of the system to create the controlling entity. This is self-limiting because for large scale distributed autonomic computing systems designing such a system based on central control becomes practically intractable. There is also a communication difficulty of providing all the relevant system information to the central controller in a timely manner. Centralised schemes require high levels of connectivity, impose a substantial computational burden, are typically more sensitive to failures than decentralised schemes, and are not scalable which is a property that distributed autonomic computing systems are required to have.

To be successful, the structure of the system has to be exploited and distributed autonomic computing systems consist of a lot of autonomous entities which directly interact with their nearest neighbours. Decentralised control then consists of using control mechanisms in software that only use local information and exert only local influences or controls in order to get a wanted effect on the global behaviour of the system. For example, in telecommunication networks, it is just not possible to gather enough information in a central node about the state of the network in a timely manner, such a central node is a single point of failure and central control is not scalable. Decentralised systems on the other hand share the decision process across the entire network. IP networks route their packets on a hop-by-hop basis with the routing decision being made for each node at each node. This is why the Internet is said to be able to recover, to be robust. If part of it is damaged or over-used then as long as the nodes neighbouring the damage are aware of this they will route packets away from the damage [39].

The main implication this has on software engineering is that we have to build software more and more with locality of information and control in mind. Every aspect of your software system is decentralised and has only limited knowledge of the other parts of the system. The big challenge here is of course to know which local influences can give rise to which global behaviours. Exploiting this knowledge in software engineering is very important to be able to achieve a completely decentralised control scheme.

### 4.4 Exploiting Nonlinearity

One of the properties of Emergence is the need for nonlinearity in the system in order for secondary effects to arise as a macro-level behaviour. Such a 'small cause, large effect' mechanism is present if a certain action is reinforced exponentially. An example is path-formation with ants (applied in [37, 24, 22]). When an ant drops a synthetic pheromone other ants will be attracted to this and when those ants reach the destination (food) they return and also drop pheromones. This reinforces the already present pheromones. In this case, the 'small cause' is that one ant lays a pheromone. This results in a 'large effect', i.e. more and more ants will start to follow and reinforce those pheromones and eventually a path is formed from the food source towards the nest.

The previous example made use of indirect communication, i.e. through the pheromones. Direct communication can also be used. When the occurrence of one specific local interaction between autonomous entities results in multiple other interactions between the same entities or other we also have a nonlinear effect. Propagation of information could be an example of this.

Emergence needs a nonlinear mechanism as an enabling mechanisms to achieve a global coherent behaviour. Due to the nonlinear reinforcement, local (inter)actions result in a larger effect in the form of an emergent at the macro-level. When engineering such systems, we have to explicitly design a nonlinear mechanism that supports the requirements.

### 4.5 The Environment as a Primary Abstraction with its own Dynamics

The importance of the environment in software engineering is formulated in the context of multi-agent systems as follows [29, 28]:

Without an environment, an agent is effectively useless. Cut off from the rest of its world, the agent can neither sense nor act. An environment provides the conditions under which an entity (agent or object) can exist. It defines the properties of the world in which an agent will function. Designing effective agents requires careful consideration of both the physical and communicational aspects of their environment.

In context-aware, open, and situated systems, autonomous entities have to incorporate the dynamics of that context in their decision making process. When engineering such software we must consider the environment or context as a primary abstraction that can have its own dynamics, independent of the intrinsic dynamics of the computing system. This is again supported by literature from the multi-agent systems community [33]: “. the dynamics of the agents interact in complex ways with those of its environment, and agent engineers need to distinguish the two”.

The dynamics of the context must be taken into account when building software for distributed autonomic computing systems. This is strongly related to the uncertainty in such distributed autonomic systems because due to the context

dynamics each action or interaction can fail. This intrinsic uncertainty is discussed in the next section.

Going even further we can exploit the context that becomes a primary abstraction as an essential coordination medium where we can engineer nonlinear coordination mechanisms to enable emergent behaviour. The environment allows indirect communication and coordination between autonomous entities. Inspired by biology, the formation of pheromone paths in ant colonies is a good example of this indirect coordination, called 'stigmergy' [11]. In general, stigmergy involves the presence of certain data in a shared context and entities in the system do certain actions as a reaction on the inspection of that shared data. These actions then influence or manipulate that data which is then again noticed and acted upon by (other) entities. This way, there is a closed feedback loop that allows nonlinearity and emergent structures to arise.

To illustrate this, we look at the use of synthetic pheromones. For example, this has been applied in Peer2Peer computing [37], Constraint Satisfaction Problems [24] and Manufacturing Control [40, 22]. This kind of stigmergy exploits the fact that the environment can have its own dynamics by requiring the evaporation of pheromones due to environmental dynamics. Evaporation means that the pheromones disappear gradually over time. This means we have two kinds of complementary feedback mechanisms. On the one hand, we have a nonlinear positive feedback by dropping pheromones where other pheromones were present. On the other hand, due to the evaporation there is a negative feedback that makes sure that old paths that are not reinforced any more will gradually disappear. As a consequence the autonomous entities that take these pheromones into account will no longer follow that path. This interplay between positive feedback from the autonomous entities and negative feedback due to the environment is a strong example of how a context-aware system can exploit the fact that it is situated in a dynamic context.

To give another example of a stigmergic system (inspired by [38, 12]) we consider a pool of bookmarks that are sorted in categories as the shared pool of data. Users execute actions on those bookmarks through a web interface. They can make their own favourites list for each category and follow interesting bookmarks by clicking on it. In response to these actions the pool of bookmarks is adjusted. For each category, less favourite bookmarks disappear while a constant number of bookmarks is kept in each category. Less favourite means 'not present in many favourites lists and not clicked very often'. This way, for each category in the pool of bookmarks more popular and more qualitative bookmarks are kept in the pool. Other users will see those qualitative bookmarks and can also visit them and/or put them in their favourites list. We have an emergence of a qualitative list of bookmarks for each category where the bookmarks themselves are the indirect communication means in a stigmergic mechanism. In addition, the lists are constantly updated so that out-dated bookmarks will eventually disappear because such bookmarks will become less popular.

In this section, we only described the environment as a indirect coordination medium. Considering the environment as

a primary abstraction can make it possible to incorporate a number of other possible mechanisms. The following mechanisms can also be important in software systems that are situated in a dynamic environment:

- Next to the dynamics of the agents there are also objects in the environment that are not autonomous but still exhibit dynamic behaviour independent from the autonomous entities.
- An environment can also explicitly represent the topology of the world in which the entities must operate to allow interactions with that world. Such interactions with the environment are called actions that the autonomous entity performs.
- A primary abstracted environment allows to incorporate rules in that abstraction that can enforce behaviour such as under which conditions actions succeed and what the effects are of conflicting actions.

These issues must also be considered in software engineering. We refer to [43] for more information on these environmental issues.

## 4.6 Intrinsic Uncertainty

The fact that an environment can have its own dynamics implies that each action or interaction performed by an autonomous entity has the possibility to fail. Autonomous entities need to be engineered in such a way that for each action or interaction there is a backup plan that is executed when failure occurs. Or, if it is not possible to detect a failure the engineer cannot assume at all that an action or interaction has succeeded. For example, in ad-hoc networking, relying on the fact that a message that is sent to a neighbouring node is always received, is unrealistic. In such networks, nodes come and go so messages sent to a leaving node will not be able to reach that node anymore. Thus, nodes whose behaviour is based on reliable communication are not flexible and robust enough.

In other words, thoroughly planning a series of actions to be performed sequentially becomes infeasible. After each action or interaction the autonomous entity should reconsider the state of the context or environment in which it is situated and act accordingly. An autonomous entity needs to be reactive, i.e. follow simple rules towards a quick response in the shape of an (inter)action that takes into account only very recent information.

Related to that is the need for an engineer not to assume that the effect of an action will always be the same. The interplay between environmental dynamics and execution of actions can give different results depending on the current state of the local environment where the action is to be executed. This means, that performing a certain (inter)action can even result in other effects than anticipated.

In short, we need to engineer systems in a way that they do not heavily rely on the normal effects of (inter)actions and do not assume that all (inter)actions will succeed. This way we achieve systems that can cope with the uncertainty

that will be intrinsically present in distributed autonomous computing systems.

#### 4.7 Engineering Good Enough, not Optimal

One could argue that the combination of locally optimal decisions of the autonomous entities often results in a sub-optimal global behaviour. It is possible that the system that is constructed is not the most optimal solution to the problem at hand. But, one of the characteristics of emergence is robustness, as described earlier. For example the failure of a single autonomous entity won't result in a complete failure of the system. And in distributed autonomous computing where we need to cope with a highly uncertain and dynamic situation this robustness is essential.

In [30] this is described as: "Robustness under continual change is worth more than a theoretical optimum in a steady-state that we never reach". P. Pirjanian and Maja J Matarić [23, 35] formulate it as follows: "Rather than searching for optimal solutions (control schemes) we seek solutions that are 'good enough', i.e. we propose to satisfice rather than optimise". In [35] the notion of optimality and its feasibility are revisited in the context of behaviour-based control. It is argued that optimal behaviour is not feasible for real-world applications. As an alternative to optimality Pareto-optimal and satisficing solutions which correspond to efficient and "good enough" behaviour are promoted.

For example, in telecommunication networks, it is less important that a packet is routed through the shortest and most optimal (fast) path. It is more important to have a very robust network that can cope with continual change (e.g. node failure, congestion, ... ).

#### 4.8 Emergence at the Right Level

When engineering systems based on emergence we have two levels of behaviour. The macro-level exhibits the global coherent behaviour of the system. The micro-level exhibits the individual local behaviour of an autonomous entity. The behaviour on each level can be characterised as being significantly different with respect to stability. The macro-level behaviour is coherent and stable. The micro-level behaviour on the other hand can be very unstable in the sense that an individual entity does not necessarily exhibit optimal and stable behaviour. Thus, it is allowed that one such entity does something that not immediately supports the wanted global behaviour. For example, an entity that walks around randomly seems to have no intention to move towards a destination. This behaviour does not directly support the task to reach destinations but it is still useful to explore the environment. It is important in software engineering to get a behaviour which we can rely on, which is stable. Therefore when engineering software we must carefully identify the level on which the emergent should be situated.

Let us clarify this with a real-world example. Consider Automated Guided Vehicles [1] for material handling. Material handling is the movement, storage, control and protection of material and goods throughout the process of their manufacture, distribution, consumption and disposal.

An automated guided vehicle (AGV) is a vehicle that is driven by an automatic control system that serves the role

of the driver. Sensors on the road or infrastructure and on-board the vehicle provide measurements about the location and speed of the vehicle which are used by the automatic control system to generate the appropriate commands for the throttle/brake actuators in order to follow certain position and speed trajectories. AGVs are considered to be a flexible type of material handling system. Their size ranges from small load carriers of a few kilograms to over 125-ton transporters. The vehicles working environment ranges from small offices with carpet floor to huge harbour dockside areas [20]. The AGV can pick up and drop off pallets or transfer loads automatically using fork attachments, conveyors, lift tops etc. depending on the type and size of the load units to handle.

The main goal for such an AGV is thus finding a good path through the environment from the pick-up point towards the drop-off point. It is not allowed for such an individual AGV to exhibit sub-optimal behaviour. For example, wandering around randomly in the wrong direction is unwanted at that level in the application. Thus, when we want to engineer an AGV-system as a distributed autonomous system so that it is scalable to large industrial transportation systems we must have a stable behaviour at the level of a single AGV. It is the emergent that is stable so the route that is followed by the AGV should be the emergent at the macro-level, a result of an emergence process. If the behaviour of a single AGV would be situated at the micro-level, like ants, that behaviour would not be stable enough.

For example, to achieve this, we can engineer a swarm of autonomous entities that constructs the paths in the environment as an emergent. That swarm can use information about obstacles, destination distance, importance of the load of an AGV, etc. The information is perceived and acted upon by the individuals in the swarm. Each AGV has the capability to send out a swarm of autonomous ants that will find the destination and lay pheromones to form the relatively stable path that an AGV can follow. This approach is used in [40] to route order through a manufacturing process. This way we combine the strength and flexibility of emergence and the requirement for a relatively optimal path for one AGV. We did not put the micro-level behaviour with the AGVs but one level lower, under the command of an AGV. The behaviour of a single AGV is situated at the macro-level.

Identifying the right level to use emergence means that the stable emergent needs to be at the level where stable behaviour is needed.

#### 4.9 Keep Autonomous Entities Simple

A major evolution outlined above is the evolution from complex entities and simple interactions towards simple entities and complex interactions. Keeping the autonomous entities simple or small has a number of implications for engineering such systems. We distinguish three dimensions in which an entity can be 'small' (inspired by [31]):

- Small in Mass: Keeping them small compared with the whole systems, i.e. not too much internal computation. Small individual entities are easier to construct

and understand than large monolithic systems, and the impact of the failure of any single autonomous entity will be minimal which is important for engineering robust software.

- **Small in Time:** Autonomous entities should be bounded in time, i.e. they are forgetful. For example, the evaporation of pheromones or a very short term memory makes it possible for the entities to adapt very quickly to changes.
- **Small in Space:** This refers to the increased locality that we discussed earlier, i.e. local sensing and acting. For example, each entity can only interact with nearby neighbours and when engineering such software these constraints should be enforced. Global effects can be obtained more robustly by propagation of local activity than by providing global activity.

Next to this simplicity inside the autonomous entities, there is of course the shift towards more complexity in the interactions, as discussed earlier.

#### 4.10 New Challenges

With these implications, no solution is given for the challenges imposed by distributed autonomic computing and emergence. They only identify new challenges with respect to software engineering for such systems. We do not state that these implications are the only ones, most likely there are more. But, they give a good overview of what we may expect when relying on emergence to get a global coherent behaviour for our systems.

### 5. STEPS TOWARDS EMERGENT SOFTWARE ENGINEERING

To achieve the vision on software engineering with emergence a lot of work has to be done. Today there is certainly not enough knowledge and support to be able to engineer emergent systems. In this section we outline what we believe to be three important steps towards emergent software engineering. First of all, we need to get a better understanding of the concept of emergence and the principles that enable it. Secondly, from that understanding we need to identify software mechanisms that can enable and be integrated in emergent-based software. And finally, towards a good software engineering methodology we need to identify the right software abstractions, architectures, tools and technologies that make it possible to efficiently engineer emergent-based software. After making enough progress in these steps, the next step is to develop a whole software development methodology and process. In this paper, we do not describe the methodology and look at the three previous steps separately.

#### 5.1 Understanding 'Emergence'

If we want to exploit emergence as a paradigm for engineering distributed autonomic computing we must first gain a profound understanding into the dynamics of the mechanisms and principles that result in emergence. If we do not have this understanding, building distributed autonomic computing systems based on emergence would be stuck in a mere trail-and-error technique.

To gain those insights there is a need for fundamental scientific methods that allow to analyse the behaviour of emergent systems. For example, in [32], the authors use a measure of entropy to analyse emergence. The fundamental claim of that paper is that the relation between self-organisation based on emergence in multi-agent systems and concepts such as entropy is not just a loose metaphor, but can provide quantitative, analytical guidelines for designing and operating agent systems. The authors explain the link between these concepts, and demonstrate by way of a simple example, i.e. synthetic pheromones, how they can be applied in measuring the behaviour of multi-agent systems. The main result is that the paper suggests that the key to reducing disorder in a multi-agent system to achieve a coherent global behaviour is coupling that system to another in which disorder increases. This corresponds to a macro-level where the order increases, i.e. a coherent behaviour arises, and the micro-level where an increase in disorder is the cause for this coherent behaviour at the macro-level.

This example shows that it can be interesting to use existing concepts and ideas from other scientific disciplines. Some other examples that have potential to result in a better understanding of emergence are chaotic time series analysis from chaos theory and analysis based on cellular automata and computational complexity theory:

- Analysing the dynamics of a nonlinear system can be done with techniques from *chaotic time series analysis*. They allow to analyse observed behaviour from measured time series which is for very complex systems often the only feasible option. Such techniques estimate characteristics defined in chaos theory and this can serve as a conceptual framework to characterise dynamical systems and explain principles of adaptive, self-organising, emergent behaviour [41].

Development of chaos theory in a specific problem domain, for example manufacturing [19], can provide a new vocabulary for understanding what happens in that system and new technologies to manage and control those systems. For more information about the possibilities of chaos theory we refer to [5].

- Stephen Wolfram describes in his 'A New Kind of Science' [44] the possibility of using *cellular automata and computational complexity theory* to analyse self-organising and emergent behaviour. In this book he argues that in computer science traditionally we engineer fairly complicated systems that yet have fairly simple behaviour to fulfill some purpose. He then states that even systems with extremely simple construction can yield behaviour of immense complexity. The behaviour of immense complexity arises from the (inter)actions between the simple entities. In this paper, this phenomenon is called 'emergence'.

Wolfram states that using geometrical and mathematical methods to characterise the possible forms of behaviour is not feasible for the behaviours of many systems that are fundamentally too complex. Another argument towards not using direct experiments that generate raw data is that they often generate huge amounts of raw data. Yet, if one manages to present

this data in the form of pictures then it effectively becomes possible to analyse very quickly just with one's eyes.

And the use of Cellular Automata makes it possible to visualise such emergent behaviour and its evolution. The specific structure of a Cellular Automaton is shown to be irrelevant for the results that can be gained. Insights can be gathered when applying his methods to the analysis of distributed autonomous computing based on emergence.

We do not state that the above fundamental scientific methods are the silver bullets for acquiring better understanding of the principles that enable emergence but future work with them can be useful to get those insights.

When we have enough insights and understanding into the process of emergence, the next step is to apply this when engineering software for such systems. In the next sections we outline two important steps for that.

## 5.2 Develop Enabling Software Mechanisms

A second important step is to do research on how we can translate the insights we got from the analysis of emergence into decentralised software mechanisms that allow to control distributed autonomous computing systems efficiently.

For example, when using chaos theory often the insights that can be gathered are related to the influence of a single variable on the global behaviour of a system. Depending on the value of that variable the system may exhibit different modes of behaviour. It can thus be expected that the analysis shows which those local influence points are. The question we then have to ask is: what control mechanisms allow us to exploit this knowledge. Learning the mapping between the entities in the system that are accessible to management and the desired influence points is important to identify which mechanisms are most effective in managing a certain influence point and thus guiding the whole system to a certain behaviour.

The development of such mechanisms in for example reusable frameworks can be a huge support for software engineers to engineer software based on emergence. They can select the mechanisms that achieve the effects they want in the global coherent behaviour. For example, a framework for implementing stigmergy based on pheromones can be very useful.

## 5.3 Support for Emergent Software Engineering

A very important step towards emergent-based software engineering is the availability of suitable support. With support we refer to a number of things. One example of it was already given in the previous section, i.e. reusable software frameworks for decentralised control mechanisms. Some other supporting steps that have to be made in future work are:

- Identifying the right software abstractions that effectively support a good and clean way of software engineering (e.g. enabling a good separation of concerns). Such software abstractions should enable properties

such as autonomy, interactions, situatedness, and uncertainty.

- Which software architectures support distributed autonomous computing systems? Which architectural styles are suited for these kind of systems? A possible idea is to combine two architectural styles:
  - *Pipe-and-filter architectures* inside the autonomous entities with the needed outputs towards the environment and other entities. Inside the environment there is also such an architecture to enable environmental dynamics.
  - *Blackboard-a-like architecture* as the shared pool of data that represents the information and entities that are present in the environment and that can be taken into account by the autonomous entities by inspection of that data.

This kind of architecture can enable the use of stigmergy-like mechanisms and makes sure that there is a very low direct coupling between the autonomous entities.

- The widespread realisation of the advantages of emergent software engineering will depend on the availability of tools for this approach, and future work should encourage the development and refinement of such tools. With respect to the software life cycle, a lot of work needs to be done in making easy to use tools that allow to analyse, design, debug and test such systems at a higher level where autonomous entity based concepts (e.g. agents) are the basic modelling entities.
- And finally, identifying or developing the right technology in order to make it easy to build such systems is crucial. As an example we can refer to the multi-agent technology that we mentioned earlier in this paper as a technology that enables a very natural mapping between the system requirements at hand and the technology to make them work.

When these supporting steps for software engineering are worked out, the next step is to develop en software methodology to engineer software based on emergence. However, this is considered outside the scope of this paper.

## 6. CONCLUSION

To summarise, this paper has given an overview of the kind of systems we must be able to engineer in the future, i.e. distributed autonomous computing systems. These systems result in a number of *shifts in the main properties of computing systems*. The most important ones are the huge increase in interaction and the fact that more complexity has to be handled autonomously (see autonomous computing). This results in very complex systems where the complexity is no longer situated in the individual entities but in the interactions between these entities. These two main shifts in the way we think about computing and some other non-traditional characteristics (i.e. situatedness and openness, locality in control and interaction, high dynamics) pose a number of challenges to cope with when engineering software for such systems. We identified these challenges as a high degree of interaction, high degree of uncertainty and

dynamics, high degree of locality, and the need for a global coherent behaviour.

The problem is to get a global coherent behaviour of your software that is constructed as a group of autonomous interacting entities. We stated that this global behaviour has to emerge from the local (inter)actions of the individual entities. As a consequence we considered the biologically inspired *paradigm of emergence*. We can conclude that engineering systems based on this paradigm, which has properties that match remarkably well on those identified for distributed autonomic computing, makes it possible to exploit and cope with the challenges imposed by such systems. To achieve emergence, locality is essential and interaction is the enabling mechanism. The intrinsic robustness of emergence can let us cope with uncertainty and dynamics. And the need for a global coherent behaviour is solved because emergents are coherent wholes that manifest themselves over time.

*Emergent Software Engineering* again poses a number of challenges to conquer. And these were described by outlining the *implications on software* when developing distributed autonomic computing systems based on emergence. A lot of work has to be done in order to achieve the goal of engineering software for distributed autonomic computing systems based on emergence. But, one thing is certain, getting inspiration from biology where emergence is the key for survival in group and applying this paradigm to computing systems is a promising approach. Certainly because there is a shift to distributed autonomic computing systems which share a lot of properties with the paradigm of emergence, and the strength and robustness in nature is needed in tomorrows computing systems.

## 7. REFERENCES

- [1] Automated guided vehicles company listing, online at <http://www.automaticguidedvehicles.com/>.
- [2] Y. Bar-Yam. *Dynamics of Complex Systems*, chapter 0, Overview: The Dynamics of Complex Systems - examples, questions, methods and concepts. Studies in Nonlinearity. Westview Press, July 1997.
- [3] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, New York, US, 1999.
- [4] J. Crutchfield. Is anything ever new? considering emergence. Working Paper 94-03-011, Santa Fe Institute, 1994.
- [5] T. De Wolf and T. Holvoet. Towards Autonomic Computing: agent-based modelling, dynamical systems analysis, and decentralised control. In *Proceedings of the First International Workshop on Autonomic Computing Principles and Architectures*, page 10, 2003.
- [6] K. Delin, S. Jackson, D. Johnson, S. Burleigh, R. Woodrow, M. McAuley, J. Britton, J. Dohm, T. Ferr, F. Ip, D. Rucker, and V. Baker. Sensor web for spatio-temporal monitoring of a hydrological environment. In *Proceedings of the 35th Lunar and Planetary Science Conference*, March 2004.
- [7] K. A. Delin, S. P. Jackson, S. C. Burleigh, D. W. Johnson, R. R. Woodrow, and J. T. Britton. The jpl sensor webs project: Fielded technology. In *Proceedings of Space Mission Challenges for Information Technology*, Pasadena, CA, July 2003.
- [8] G. Ford. SEI Report on Undergraduate Software Engineering Education. Technical Report CMU/SEI-90-TR-003, SEI, 1990.
- [9] P. Giorgini, J. P. Mller, and J. Odell. Call for Papers for the The Fifth International Workshop on Agent-Oriented Software Engineering. held at the The Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2004), New York City, New York - July 19, 2004, 2004.
- [10] J. Goldstein. Emergence as a construct: History and issues. *Emergence*, 1(1), 1999.
- [11] P. Grasse. La reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. la theorie de la stigmergie: Essai d'interpretation des termites constructeurs. *Ins. Soc.*, 6:41–83, 1959.
- [12] J. Gregorio. Stigmergy and the World-Wide Web. online at <http://bitworking.org/news/Stigmergy>, December 2002.
- [13] A. Helleboogh, D. Weyns, and T. Holvoet. Time management adaptability in multi-agent systems. In *Proceedings of the Fourth International Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS-4)*, 2004. (accepted).
- [14] F. Heyligen. Self-organization, emergence and the architecture of complexity. In *Proceedings of the 1st European Conference on System Science*, Paris, 1989.
- [15] F. Heyligen. The science of self-organisation and adaptivity. In *The Encyclopedia of Life Support Systems*. UNESCO Publishing-Eolss Publishers, Oxford, UK, 2002.
- [16] J. Holland. *Emergence: from Chaos to Order*. Addison-Wesley, 1998.
- [17] M. N. Huhns. Interaction-Oriented Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):259–279, 2001.
- [18] IBM. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. IBM, 2001.
- [19] Y. Indrayadi. *Distributed Dispatching Control For Dynamic Flow-Line Manufacturing Systems*. PhD thesis, Katholieke Universiteit Leuven, September 2002.
- [20] P. Ioannou, H. Julia, C.-I. Liu, K. Vukadinovic, H. Pourmohammadi, and J. Edmond Dougherty. Advanced material handling: Automated guided vehicles in agile ports. Technical report, Center for Commercial Deployment of Transportation Technologies, University of Southern California, 2001. (available at [http://www.usc.edu/dept/ee/catt/..../2002/jula/Marine/FinalReport\\_CCDoTT\\_981.pdf](http://www.usc.edu/dept/ee/catt/..../2002/jula/Marine/FinalReport_CCDoTT_981.pdf)).

- [21] N. Jet Propulsion Laboratory. Nasa/jpl sensor webs project. online at <http://sensorwebs.jpl.nasa.gov/>.
- [22] H. Karuna, P. Valckenaers, C. Zamfirescu, H. Van Brussel, B. Saint-Germain, T. Holvoet, and E. Steegmans. Self-Organising in Multi-Agent Coordination and Control Using Stigmergy. In *Proceedings of the First International Workshop on Engineering Self-Organising Applications*, 2003.
- [23] M. J. Matarić. *Interaction and Intelligent Behavior*. PhD thesis, Department of Electrical Engineering and Computer Science at Massachusetts Institute of Technology (MIT), May 1994.
- [24] K. Mertens, E. Steegmans, and T. Holvoet. Cyclic Path-Based Environment: an Ant Environment for Solving Distributed Constraint Satisfaction Problems. In *Proceedings of the Third International Workshop on Distributed Constraint Reasoning*, pages 94–103, 2002.
- [25] J. Odell. Agents and complex systems. *Journal of Object Technology*, 1(2):35–45, July-August 2002.
- [26] J. Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, May-June 2002.
- [27] J. Odell, H. V. D. Parunak, and B. Bauer. Representing Agent Interaction Protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, page 121?140, Berlin, 2001. Springer-Verlag. (Held at the 22nd International Conference on Software Engineering (ISCE)).
- [28] J. Odell, H. V. D. Parunak, and M. Fleischer. Modeling agents and their environment: The communication environment. *Journal of Object Technology*, 2(3):39–52, May-June 2003. [http://www.jot.fm/issues/issue\\_2003\\_05/column5](http://www.jot.fm/issues/issue_2003_05/column5).
- [29] J. Odell, H. V. D. Parunak, M. Fleischer, and S. Brueckner. Modeling agents and their environment: The physical environment. *Journal of Object Technology*, 2(2):43–51, March-April 2003. [http://www.jot.fm/issues/issue\\_2003\\_03/column5](http://www.jot.fm/issues/issue_2003_03/column5).
- [30] H. D. V. Parunak. Tutorial: Self-organizing behavior at the fourth international conference and exhibition on the practical application of intelligent agents and multi-agent technology (paam'99). slides available at <http://www.erim.org/~vparunak/>, 1999.
- [31] H. V. D. Parunak. “go to the ant”: Engineering principles from natural agent systems. *Annals of Operation Research*, 75:69–101, 1997. (available at <http://www.erim.org/vparunak/>).
- [32] H. V. D. Parunak and S. Brueckner. Entropy and self-organization in multi-agent systems. In J. P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 124–130, Montreal, Canada, 2001. ACM Press.
- [33] H. V. D. Parunak, S. Brueckner, J. Sauter, and R. S. Matthews. Distinguishing Environmental and Agent Dynamics: A Case Study in Abstraction and Alternate Modeling Technologies. In *Proceedings of International Workshop on Engineering Societies in the Agents' World at ECAI'00*, 2000. (available at <http://www.erim.org/~vparunak/>).
- [34] V. Parunak and R. VanderBok. Modeling the extended supply network, 1998.
- [35] P. Pirjanian. Multiple objective behavior-based control. *Robotics and Autonomous Systems*, 31(1-2):53–60, April 2000.
- [36] J. A. Sauter and H. V. D. Parunak. Ants in the supply chain. In *Proceedings of Workshop on Agent based Decision Support for Managing the Internet-Enabled Supply Chain at Agents 99*, May 1999.
- [37] K. Schelfhout and T. Holvoet. A pheromone-based coordination mechanism applied in P2P. In *Proceedings of the Second International Workshop on Agents and Peer-to-Peer Computing*, pages 151–162, 2003.
- [38] P. Small. Stigmergic systems. online. (example application available at <http://www.stigmergicsystems.com>).
- [39] J. Spencer. Complexity in broadband networks with decentralised control. Master's thesis, Dept. of Electronic and Electrical Engineering University, College London,, London, England, 1997/98.
- [40] E. Steegmans, T. Holvoet, N. Janssens, S. Michiels, Y. Berbers, P. Verbaeten, P. Valckenaers, and H. Van Brussel. Ant Algorithms in a Graph Environment: a Meta-scheme for Coordination and Control. In *Artificial Intelligence and Applications*, pages 435–440. International Association of Science and Technology for Development - IASTED, ACTA Press, 2002.
- [41] H. Van Parunak. Complexity theory in manufacturing engineering: Conceptual roles and research opportunities. Technical report, Industrial Technology Institute, 1993. (available at <http://www.erim.org/~vparunak/>).
- [42] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, May 1997.
- [43] D. Weyns and T. Holvoet. Formal model for situated multi-agent systems. *Formal Approaches for Multi-agent Systems, Special Issue of Fundamenta Informaticae*, to appear in 2004.
- [44] S. Wolfram. *A New Kind of Science*. Wolfram Media, Inc., 2002. (available at <http://www.wolframscience.com/nksonline/>).
- [45] F. Zambonelli and H. V. D. Parunak. Signs of a revolution in computer science and software engineering. In *Engineering Societies in the Agents World III, Third International Workshop, ESAW 2002, Madrid, Spain, September 16-17, 2002, Revised Papers*, volume 2577 of *Lecture Notes in Computer Science*. Springer, 2003.