From Design to Intention: Signs of a Revolution

Franco Zambonelli
Dipartimento di Scienze dell'Ingegneria – Università di Modena e Reggio Emilia
Via Vignolese 905 – 41100 Modena – ITALY
Phone: +39-059-2056133 – Fax: +39-059-2056126
franco.zambonelli@unimo.it

Abstract

In this paper, we identify and analyze a set of issues that are more and more influencing the characteristics of today's complex software systems, and that make them distinguish from "traditional" software systems. Several examples in different areas show that these issues do not influence a few application domains only, but are instead widespread. Then, we discuss how these issues are likely to dramatically impact on the very way software is modeled and engineered. In particular, we show that we are on the edge of a revolutionary shift of paradigm, likely to change our very attitudes, and making us conceive software systems no longer in terms of mechanical systems, but rather in terms of intentional or physical systems.

1 Introduction

Computer science and software engineering are going to dramatically change. Scientists and engineers are spending a great deal of efforts in the attempt of continuously adapting and improving well-established models and methodologies for software development. However, the complexity raised in software systems by several emerging computing scenarios seems to be no longer affordable in terms of the abstractions and methodologies promoted by traditional approaches to computer science and software engineering, such as object-oriented and component-based ones.

The so called "software crisis" occurred in the 70's and having led to modular and object-oriented programming paradigm was mainly caused by a problem of size. The required amount of functionality in software systems were increasing over the limits to be effectively handled by adopting an "algorithmic" approach to software modeling and an "artisan-based" approach to software development, and led to an "architectural" approach to software systems design and modeling. The same driving force has primarily driven research and technological efforts till today, leading to modern approaches to object-oriented and component-oriented programming. The crisis that we are going to face in the next few years will be of a very different nature, and the never stopping increase in size of software systems will represent only a single facet of it.

The scenario that will cause the next software crisis is rapidly forming under the eyes of everybody: computing systems are going to be everywhere and will be always connected and active [Ten00]. On the one hand, computer systems are going to be embedded in every object, e.g., in our everyday clothes, in our home furniture, in our homes themselves, not to mention the computing capabilities of handheld computing devices and cellular phones, which are already a reality. On the other hand, also due to the advances in wireless technologies, network connectivity will be pervasive, and every computing device will be somehow connected in a network, whether the "traditional" Internet or some ad-hoc network. The diffusion of WAP and of the incoming UMTS technology, as well as the possibility of accessing the Internet via electricity grids, are nothing but the first embryonic stage of this trend. Finally, computing systems will always "do something", i.e, will be always active to perform some activity on our behalf, let these activities be devoted to improve our comfort at home or to control and automate manufacturing processes in industries. After all, daemons and assistant agents already populates our PCs and workstations, devoted to monitor and handle our computing and digital resources.

In this paper, we argue that the above scenarios do not simply quantitatively affects – in terms of number of components and effort required – the design and development of software systems. Instead, we argue that there will be a qualitative change in the characteristics of software systems, as well in the methodologies adopted to develop them. In particular, we argue that 4 main characteristics – in addition to the quantitative increase in interconnected computing systems – distinguish future software systems from traditional ones: situatedness, openness, locality in control, and locality in interactions.

Any researcher having worked in the area of agent-based computing will immediately recognize the above characteristics as the main distinguishing ones of agents and of multi-agent systems. However, in this paper we go into details about the above four characteristics and will try to show how, to different extents and with different terminology, several research communities are already recognizing their importance and are already adapting their models and technologies to take them into account. Thus, a first contribution of this paper is attempting at synthesizing in a single conceptual framework a multitude of novel

concepts and abstractions that – often because of a lack of interaction and common terminology between research groups – are emerging in different areas without recognition of the basic commonalties.

Following the above synthesis attempt, the paper argues that the integration of the above concepts and abstractions in software modeling and design does not simply represent a proper evolution of current models and methodologies. Instead, the paper claims that we going to pass though a real revolution, a radical change of paradigm [Kuh96]. Such a revolution will dramatically change the very way we conceive and model "software components" and "software systems", as well as the way we will build such systems. In a sentence, the paper argues that next generation software systems will no longer be modeled and designed in terms of "mechanical" or "architectural" systems, but rather in terms of "physical" or "intentional" systems. Accordingly, the paper tries to identify the impact of such a change of paradigm in computer science and software engineering practices.

2 What's New?

In this section, we go into details about the four characteristics identified in the introduction, namely situatedness, openness, local control, local interactions, and shows how they affect most of today's software systems, other than agent-based ones.

2.1 Situatedness

Today's computing systems are situated. That is, they have an explicit notion of the environment in which components are allocated and execute, are affected in their execution by the environmental characteristics, and their components often explicitly interact with that environment.

We emphasize that software systems have never worked in isolation, but have always been and will always be immersed in some sort of environment. For instance, a running process in a multi-threaded machine is intrinsically affected in its execution by the dynamics of the multiprocessing system. However, traditional modeling and engineering approaches have always tried to mask the presence of the environment. In most of the cases, specific objects and components were devoted to "wrap" the environment and to model it in terms of a "normal" component. That is, the environment in itself did not exist as a primary abstraction in approaches to software development. Unfortunately, the environment in which components live and with which they interact indeed exists and may impact on application execution and on application modeling:

- several entities with which software components may be in need of interacting are too complex in their structure and behavior to enable a trivial wrapping;
- for a software system whose explicit goal is to monitor (sense) and control (affect) a physical environment or a logical – software – environment masking such an environment seems simply not a natural choice;
- the environment can have its own dynamics, independent of the intrinsic dynamic of a software systems. Wrapping the environment into an application entity, will introduce forms of unpredictability non-determinism in the behavior of some parts of our applications.

Due to the above reasons, the current trend in both computer science and software engineering is to explicitly elect the environment in which a software system executes as a primary abstraction. This implies explicitly defining both the "boundaries" separating the software systems and its environment and the reciprocal influences of the two systems. On the one hand, this avoids odd wrapping activities needed to model each component of the external world as an internal application entity. On the other hand, this allows a software system to deal in a more natural with its activities on a real-world environment it is devoted to monitor and control. In addition, the explicit modeling of the environment and of its activities enable to clearly identify and confine the sources of dynamicity and unpredictability, and concentrate on our software components in terms of deterministic entities that have to deal with a dynamic – possibly unpredictable – environment.

Examples. There are several notable examples of this trend. Systems for the control of manufacturing systems and, more generally, embedded systems for the control of physical environments (traffic control systems, home care and health care system) tends to be built by explicitly managing environmental data, and by explicitly taking into account the unpredictable dynamics of the environment via specific event-handling policies. Internet applications and web-based systems, to be dived into the existing and intrinsically dynamic Internet environment, are typically engineered by clearly defining the boundaries of the system in terms of "application-level", including the new application components to be developed, and "middleware" level, as the environmental substrate in which components will be dived. Even more, clearly identifying and defining such boundaries is one of the key point in web-application engineering. As a further example, several systems for workflow management and computer supported collaborative work are built around shared data spaces abstractions, to be exploited as the common environment for the execution of workflow processes and agents. As a final example, it is worth noting that several promising proposals in the area of distributed problem solving and optimization (i.e., works on ant-based colonies [Par01]), are based on a model centered around a dynamic virtual environment influencing the activities of distributed problem solver processes.

2.2 Openness

Living in an environment, perceiving it, and being affected by intrinsically implies some sort of openness of a software system. In fact, the system cannot be longer conceived as an isolated world, but it must be instead considered as permeable sub-system, through which boundaries reciprocal side-effects may occur. However, this form of reciprocal influence system-environment often assume extreme and very complex forms, making it difficult to clearly identify boundaries between the systems and its environment.

In several cases, software systems are explicitly designed to interact with external software components, whose functionality and characteristics are required for the system in itself to achieve the objectives it is designed for. From the opposite perspective, the component of a software system may be designed to be involved in interactions with other components, to provide them services or data. More in general, different software systems – independently designed and modeled – are likely to "live" in the same environment and explicitly interact with each other. These forms of open interactions call for common ontologies, communication protocols, and suitable broker infrastructures, to enable interoperability. However, this is only a small part of the problem.

Simply enabling interoperability is not enough when software system may come to life and may death in an environment independently of each other, in a very dynamic way, or when a software system (or a part of it) can explicitly move across different environments during its life. These characteristics introduce additional problems:

- a component getting to life in an environment (or having somehow moved to a specific environment) requires somehow being made aware of what's around in the environment, and of what other components are there for interaction;
- when a component interacts with other components in other software systems, and when components move across different environment, it may become hard if not impossible clearly identifying to which system a component belongs to or – which is the same – it becomes difficult to clearly understand the boundaries of software systems;
- allowing components to enter and leave an environment in a fully free way, and interact with each other may make it very hard to understand and control the overall behavior of these software systems and even of a single software system.

Due to the above problems, the current trends in computer science and software engineering is to start considering the problem of not only modeling the environment in which systems execute, but also of the context in which they execute. In fact, the aim is also to understand the way software components can be made somehow aware of the context in which they execute, and the way interaction of independent components in an open context con be controlled and ruled.

Examples. Let us consider those control systems for critical environments, such as traffic control systems, public telephony services, health care systems, as well as manufacturing systems. All this systems are forever running, cannot be stopped, and sometimes cannot even be removed from the environment in which they are embedded. Nevertheless, this systems need to be continuously updated, and the environment is which they live is likely to change frequently, with the addition of new physical components and, consequently, of new software components and software systems. For all these systems, managing openness and the capability of establishing new effective interactions with new components is of dramatic importance, as it is important the capability of a component of safety and effectively entering new execution contexts. Moreover, it is important that execution contexts are able to somehow constraints and control the dynamic interactions there occurring. Very similar considerations apply to Internet based and open distributed computing. There, software services must survive the dynamics and uncertainty of the Internet, must be able to serve any client component, and must be as well able to enact security and resource control policy in their local context. As a final example in which openness and context plays a very important role relates to mobility. Mobility, whether of users, of software, or of devices, moves the concept of openness at the extreme, by making components actually move from one context to another during their execution [Whi97, CabLZ01]. Of course, more than in other application areas, this requires the capability of components of dynamically acquiring knowledge about the context in which they execute, and the capability of controlling component interaction in a context.

2.3 Local Control

The "flow of control" concept has always been one of the key aspect of computer science and software engineering, at all levels, from the hardware level up to the high-level design of applications. However, when software systems and components live and interact in an open world the concept of flow of control simply lose any meaning.

Independent software systems have their own autonomous flows of control, and their mutual interactions do not imply any join of this flows. Therefore, the modeling and designing of open software not only makes the concept of "software system" rather vague, as discussed in Subsection 2.2, but it also make the concept of "global execution control" disappear.

This trend is exacerbated by the fact that not only independent systems have their own flow of control, but also different components in a system may be autonomous in

control. In fact, most components of today's software systems are active entities (e.g., active objects with internal threads of control, processes, daemons) rather than passive one (e.g., "normal" objects, functions, etc.).

We are aware that having multiple threads of control in a system is not a big novelty, and that concurrent and parallel programming are here since a long time ago. However, efficiency and performance mainly drove traditional concurrent and parallel programming in introducing multiple threads of control in applications. Apart from that, most of the approaches ion the area aimed at limiting as much as possible the independence of these multiple threads of control, via strict synchronization and coordination mechanisms, with the goal of preserving determinism and high-level control over applications. Today's autonomy of application components is here for different reasons and has to be faced in very different way:

- in an open world, autonomy of execution facilitates a components in moving across systems and environments without having to report back to (or wait for ack by) its original application;
- when components and systems are immersed in a highly dynamic environment, and there is need of monitoring and controlling the evolution of the environment, an autonomous components can be effectively delegated of taking care of (a portion of) the environment independently of the global flow of control;
- several software systems are not only made up of software components, but integrates also computer-based systems, which are for their very nature autonomous systems, and can be modeled accordingly;
- as the dimension of a software system increases, the need of delegating control to components is no longer simply a matter of performance, but it becomes a matter of conceptual simplicity. In fact, coordinating a global flow of control within a very large number of components may become unfeasible. Autonomy then becomes an additional dimension of modularity, naturally extending the concepts of autonomy in data management promoted by object-oriented computing [Par97].

It is worth outlining that our concept of autonomy encompasses the concept of autonomy usually adopted in the area of agent-based computing, in that it refer not only to those software components that are explicitly designed as autonomous, but also to those components that can be perceived as autonomous.

Examples. Almost all of today's software systems integrates autonomous components. At weakest, autonomy reduces to the capability of a component of reacting to and handling events, as it can be the case of graphical interfaces or simple embedded sensors. However, in many cases, autonomy implies that a component integrates an autonomous thread of execution, and can execute in a proactive way. This is the case of most of modern control systems, in which control is not simply reactive but proactive, realized via a set of cooperative autonomous processes or, as it is often the case, via embedded complete computer-based systems interacting with each other. Internet based distributed applications are typically made up of autonomous processes, possibly executing on different nodes, and cooperating with each other, a choice driven by conceptual simplicity rather than by the actual need of autonomous concurrent activities. Finally, the integration in distributed applications and systems of mobile devices, which are autonomous computing systems, can be tackled only by modeling them in terms of autonomous software components.

2.4 Local Interactions

Directly deriving from the above three issues, the concept of "local interactions in a global world" is more and more pervading today's software systems.

By now, we have delineated a scenario in which software systems components are immersed in an specific environment, execute in the context of a specific (sub)system, and are delegated of performing some task in autonomy. Taken all together, the above aspects naturally lead to a strict enforcement of locality in interactions. In fact:

- autonomous component can interact with the environment in which they are immersed, by sensing and effecting it. If the environment is of a physical nature, the amount of physical world a single component can sense and effect is locally bounded by physical laws. If the environment is a logical one, minimization of conceptual and management complexity suggest in any case modeling it in terms of a locality and limiting the effect of a single component on the environment;
- components can normally interact with each other in the context of the software system they belong to, that is, locally to their system. In open work, however, a component of a system can also interact with (components of) another systems. In these cases, minimization of conceptual complexity suggest modeling the component in terms of a component that has temporarily "moved" to another context, and that interact locally in the new context [CabLZ01];
- in an open world, components need some form of context-awareness to execute and interact effectively. However, for a component to be effectively (and efficiently) made aware of its context, this context must be necessarily locally confined.

As a final note, it is worth outlining that it is well known that locality in interactions is a strong requirement when the number of components in a system increases, or as the

dimension of the distribution scale increases. In any case, it is worth outlining that the presence of autonomous threads of control may make tracking and controlling concurrent, autonomous, and autonomously initiated interactions much more difficult than in object-based and component-based applications, even if these interactions are strictly local.

Examples. Control and manufacturing systems tend to enforce local interactions because of their very nature. Each control component is delegated to control a portion of the environment, and its interactions with other components are usually limited to those ones that control neighboring portions of that environment, with which it typically has strict coordination requirements. Applications distributed in the Internet have to take into account the specific characteristics of the local administrative domain in which its components execute and have to interact, and components are usually allocated in Internet nodes so as to enforce as much as possible locality in Interactions. In mobile computing applications, there included applications for wearable systems, the very nature of wireless connections forces locality in interactions. Being wireless communication capabilities of limited range, a mobile computing component can directly interact – at a given time – only with a limited portion of the world.

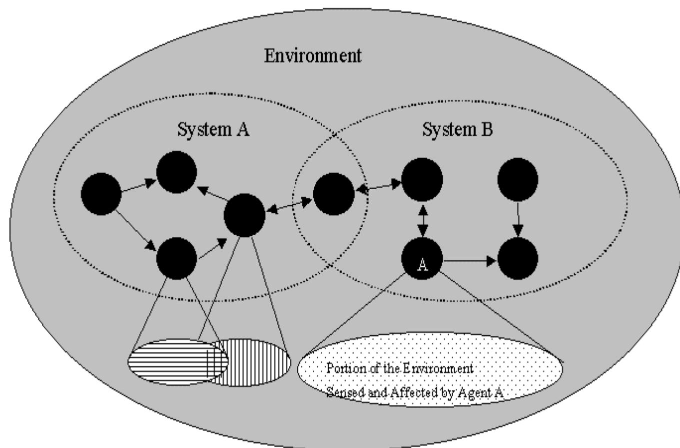

Fig. 1. The Scenario of Modern Software Systems

2.5 A General Model

Summarizing, software systems can be more and more assimilated to the general scheme of Figure 1.

Software systems (dashed ellipses) are made up of autonomous components (black circle), interacting locally with each other and possibly with the components of other systems. Accordingly, some components may though as be part of several software systems at different times, depending on their current interaction activities.

Systems and components are immersed in an environment, typically composed by (or modeled as) a set of different environment partitions. Components in a system can sense and effect a local portion of the environment. Also, since the portions of the environment that two components may access may overlap with each other, two components may interact indirectly with each other via the environment. From the point of view of components, however, this form of interaction is perceived in terms of environmental dynamics.

It is worth noting that the figures depict a scenario that is very different from the one of component-based and object-based programming. Here, the notion of "application" is substituted by the notion of "interacting systems". Here, not only objects and components are part of the system, but the environment too – and its not controllable dynamics – are first order abstractions. Here, the contractually specified and typically static patterns of interaction of object-based programming are substituted by dynamic and autonomously initiated interaction patterns. Dynamic interactions also include those of a component with the environment and the indirect interactions between components that can occur via the environment.

The outlined scenario is the one that has already been largely adopted as the basic model of agent-based computing [Jen01]. In fact, agents are considered as situated software entities (that is, that live dipped in an environment) that executed in autonomy (i.e., have local control on their actions) and that interact with other agents (local interactions are often promoted, although they are not explicitly mentioned as part of the model). Despite this fact, "traditional" computer scientist and software engineers do not admit that they are already doing system that can be assimilated to agent-based computing and modeled as agent-based systems. Neither most of the scientist working on agent-based computing recognize that agents and agent-based computing have to potential to be a general model for today's computing system: the general opinion is that if agents are not intelligent, they are nothing.

It is our hope that the issues discussed in this paper can have somehow helped clarifying these strict relationships.

3 Changing our Attitudes

Till now, software systems have been modeled by adopting a mechanical attitude, and engineered by adopting a design attitude. On the one hand, computer scientists are both burdened and fascinated by the urge of defining suitable formal theories of computation, of proving properties of software systems, and of providing formal framework to the use of engineering. On the other hand, software engineers are used at analyzing the functionality that a system should exhibits, and at designing software architectures as reliable multi-component machines, capable of providing the required functionality in an efficient and predictable way.

In the future, scientist and engineer will have to design software system so as to execute in a world where an uncountable multitude of autonomous, embedded, and mobile software components, are already executing and

interacting with each other and with the environment on the basis of local interaction patterns. However, such a scenario may require different models and methodology to be faced than the traditional ones.

3.1 How Will Computer Science Change?

Modeling and handling a very large number of components can be feasible is such components are not autonomous, i.e., are subject to a single flow of control. However, when the activities of these components are autonomous, it is hard – if not impossible – to track them one by one and, so, to precisely describe the behavior of a system in terms of the behavior of its components. Instead, all that can be done in these cases is describing and modeling the system as a whole, in terms of macro-level characteristics, the same as a chemist describes the characteristics of a gas in terms of properties like pressure and temperature.

The above problem is exacerbated by the fact that components will interact with each other. Accordingly, the overall behavior of a system will emerge not only as the sum of the behavior of its components, but also as a result of their mutual interactions. One may argue that since interactions tends to be usually confined within a locality, as explained in Subsection 2.4, this should not represent a big problem and, instead, it should be quite easy to control the effect of interactions. Unfortunately, this is not the case, and the effect of local interactions in a global world can be very difficult to predict. In fact, a large number of components locally interacting with each other can produce global macro scale effects on the behavior of a system, and these effects can be impossible to be evaluated without knowing exactly the initial status of the system [PriS91].

As a final additional problem, we must consider that software systems will execute in an open and dynamic environment, that new components can be added and removed at any time, and that some of these components can autonomously move from one part of the other of the system. Thus, not only the behavior of the system is difficult to be exactly predicted and controlled, but it is also very hard to predict how such behavior can be influenced by external driving forces, such as of the environmental dynamics. Modern thermodynamic, as well as modern social sciences, tells us that that environment forces can produce very strange and large scale behaviors on open system. Thus, we can expect the same large-scale effects to emerge on situated and open software systems.

Taken all together, the above problems will force computer scientist to dramatically change their attitudes in the modeling of complex software systems. The dream of building fully formalizing systems has already started to vanish the advent of concurrent and interactive systems [Web97], and it will definitely disappear in the next few years. The next challenge is to find alternative models or – more radically – to adopt a brand new scientific background for the study of software system.

Some signals of this trend can already be found in different areas and research community. Recent study and monitoring activities on the Internet have made it clear that unpredictable and large-scale behaviors are already here, requiring new models and tools to be described [CroB96]. Some approaches to model and describe software systems in terms of thermodynamic systems have already been proposed [ParB01]. Not surprisingly, one of the areas in which such trend is highly evident is modern artificial intelligence. The concept of "rational" intelligence, as it should have emerged from a complex machine capable of manipulating facts and logic theories, is being abandoned. Now, the abstractions promoted by agent-based computing (well matching the characteristics of today's software systems, as explained in Subsection 2.5) have put emphasis on the concept of "intentional" intelligence, i.e., the capability of a component or of a system to behave in autonomy so as to achieve a given goal. Organizational [Zam01] and social science [MosT95] are starting influencing research works, in that it is recognized that the behavior of large scale software system – rather than to a logic or mechanical system – can be assimilated to a human organization aimed at reaching a global organizational goal, or to a society in which the overall global behavior derives from the self-interested intentional behavior of its individual. In addition, the complex mechanisms of biological ecosystem are more and more providing inspiration to computer scientist [HubH93].

In the future, we expect theories and models from complex dynamical systems, from modern thermodynamics, as well as from biology, social science, and organizational science, to become the sine-qua-non cultural background of computer scientist.

3.2 How Will Software Engineering Change?

The change in the modeling and understanding of complex software systems will also dramatically impact in the way such systems are designed, maintained, and tested.

By now, software systems are designed so as to exhibit a specific, predictable, and deterministic behavior at any level of the software system: from the level of single units up to the level of the whole systems. The next challenge for the effective building of large software systems, overcoming the impossibility of controlling the behavior of each of its single components and their interactions, is to build it so as to (reasonably) guarantee that the system will behave as desired despite the exact knowledge about its micro-behavior. For instance, by adopting a "physical" attitude on software design, a possible approach could be to build a system that, despite the uncertainty on the initial conditions, is able to reach a given stable basin of attraction. By adopting a "teleological" attitude, the idea could be to build an ecosystem, or a society of components, able to behave in an

intentional way, and robust enough to direct its global activities toward the achievement of the required goal.

Again, it is possible to identify a few works that are already adopting such a novel software engineering perspective. In the area of distributed operating systems management, policies for the management of distributed resources are already being designed in terms of autonomous components able to guarantee the achievement of a global goal via local actions and local interactions [Cyb89]. Cellular automata can be made evolve so as to exhibit useful global behaviors without anyone having directly programmed such behaviors [Sip99]. Systems of ant-colonies are shown to be able to solve very complex problems via the interactions of very simple autonomous components [Par97].

In addition to the change in the way software is designed, the new scenario will also dramatically impact in the way software is tested and maintained, and evaluated.

A large software system will be no longer tested with the goal of finding errors in it, but it will be rather tested with regard to its capability of behaving as needed as a whole, independently of the exact behavior of its components and of their initial conditions [Huh01]. Moreover, a software system that is likely to be dived in an existing dynamic environment, where other systems are already executing and cannot be stopped, cannot be simply tested and evaluated in terms of its capability of achieving the required goal. Instead, the test must also evaluate the effect of the environment on the software system, as well as the effects of the software system on the environment. The better and more robust a system, the higher its capability of acting towards its goals despite the dynamics of the environment.

Maintaining software will change too. When a large software system does no longer behave as needed, as when the external condition will require some change in the behavior of a software system, update will imply no longer stopping the system, re-build it, and re-testing it. Instead, it will imply intervening on the system from the external, by adding new components with different attitudes and by removing some of its existing components so as to change the overall behavior of the system as needed.

4 Conclusions

Modern software systems, in different application areas, exhibit characteristics that make them very different from the software systems that we, as scientists and engineers, are used to deal with. These characteristics are likely to dramatically impact on the very way software systems will be modeled and engineered, leading to a true revolution in computer science and software engineering [Kuh96]. In fact, we will be required to change our traditional "design" attitude, leading to a mechanical perspective, into an "intentional" attitude, leading to physical, biological, and teleological perspectives. Despite the opposing forces and the difficulties inherent in any revolutionary phase, there included the need of re-structuring our cultural background, this revolution will definitely open up the door for new interesting research and engineering challenges.

References

[CabLZ01] G. Cabri, L. Leonardi, F. Zambonelli, "Engineering Mobile Agent Applications via Context-Dependent Coordination", *23rd International Conference on Software Engineering*, ACM, Toronto (CA), May 2001.

[Cap97] F. Capra, *The Web of Life: The New Understanding of Living Systems*, Doubleday, Oct. 1997.

[CroB96] M. Crovella, A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Causes" *ACM Sigmetrics*, pp. 160-169, 1996.

[Cyb89] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors", *Journal of Parallel & Distributed Computing*, 7(2), Feb. 1989.

[HubH93] B. A. Huberman, T. Hogg, "The emergence of computational ecologies", in *Lectures in complex systems*, Addison-Wesley, 1993.

[Huh01] M. Huhns, "Interaction-Oriented Programming", *1st International Workshop on Agent-Oriented Software Engineering*, LNCS No. 1957, Jan. 2001.

[Jen01] N. R. Jennings, "On Agent-Based Software Engineering", *Artificial Intelligence*, 117(2), 2000.

[Kuh96] T. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, 3rd Edition, Nov. 1996.

[MosT95] Y. Moses, M. Tenneholtz, "Artificial Social Systems", *Computers and Artificial Intelligence*, 14(3):533-562, 1995.

[ParB01] V. Parunak, S. Bruekner, "Entropy and Self-Organization in Agent Systems", *5th International Conference on Autonomous Agents*, ACM Press, May 2001.

[Par97] H. V. D. Parunak, "Go to the Ant: Engineering Principles from Natural Agent Systems", *Annals of Operations Research*, 75:69-101, 1997.

[PriS91] I. Prigogine, I. Steingers, *The End of Certainty: Time, Chaos, and the New Laws of Nature*, Free Press, 1997.

[Sip99] M. Sipper. The Emergence of Cellular Computing. *IEEE Computer*, 37(7):18-26, July 1999.

[Ten00] D.Tennenhouse, "Proactive Computing", *Communications of the ACM*, May 2000.

[Weg97] P. Wegner. "Why Interaction is More Powerful than Computing", *Communications of the ACM*, 1997.

[Whi97] J. White, "Mobile Agents", in *Software Agents*, AAAI Press, Menlo Park (CA), pp. 437-472, 1997.

[Zam01] F. Zambonelli, N. R. Jennings, M. J. Wooldridge, "Organizational Abstractions for the Analysis and Design of Multi-agent Systems, *1st International Workshop on Agent-Oriented Software Engineering*, LNCS No. 1957, Jan. 2001.