

Signs of a Revolution in Computer Science and Software Engineering

Franco Zambonelli¹, H. Van Dyke Parunak²

1) Dip. di Scienze e Metodi dell'Ingegneria – Univ. di Modena e Reggio Emilia
franco.zambonelli@unimo.it

2) Altarum - 3520 Green Ct, Suite 300, Ann Arbor, MI 48105 USA
van.parunak@altarum.org

Abstract. *Several characteristics distinguish today's complex software systems from "traditional" ones. Examples in different areas show that these characteristics, already the focus of agent-oriented software engineering research, influence many application domains. These characteristics will impact how software systems are modeled and engineered. We are on the edge of a revolutionary shift of paradigm, pioneered by the multi-agent systems community, and likely to change our very attitudes in software systems modeling and engineering.*

1 Introduction

Computer science and software engineering are going to change dramatically. Scientists and engineers are spending a great deal of effort attempting to adapt and improve well-established models and methodologies for software development. However, the complexity introduced to software systems by several emerging computing scenarios goes beyond the capabilities of traditional computer science and software engineering abstractions, such as object-oriented and component-based methodologies.

The scenario initiating the next software crisis is rapidly emerging: computing systems will be everywhere, always connected, and always active [Ten00]. Computer systems will be embedded in every object, e.g., in our physical environments, our clothes and furniture, and even our bodies. Wireless technologies will make network connectivity pervasive, and every computing device will be connected in some network, whether the "traditional" Internet or ad-hoc local networks. Finally, computing systems will be always active to perform some activity on our behalf, e.g., to improve comfort at home or to control and automate manufacturing processes.

This scenario does not simply affect the design and development of software systems *quantitatively*, in terms of number of components and effort required. Instead, there will be a *qualitative* change in the characteristics of software systems, as well as in the methodologies adopted to develop them. In particular, four main characteristics, in addition to the quantitative increase in interconnected computing systems, distinguish future software systems from traditional ones:

- *situatedness*: software components execute in the context of an environment, can influence it, and can be influenced by it;
- *openness*: software systems are subject to decentralized management and can dynamically change their structure;

- *locality in control*: the components of software systems represents autonomous and proactive loci of control;
- *locality in interactions*: despite living in a fully connected world, software components interact accordingly to local (geographical or logical) patterns.

These characteristics commonly characterize multi-agent systems (MAS) [Jen01]. Section 2 discusses them characteristics in detail and shows how, to different extents and with different terminology, various research communities (e.g., manufacturing control systems [Bus00], mobile and pervasive computing [Wei93, AboM00], sensor networks [Est02], groupware and enterprise infrastructures [MamLZ02, Tol00], Internet [CabLZ02] and P2P computing [PieIF02]) already recognize their importance and are adapting their models and technologies to them. Thus, our first contribution is to synthesize in a single conceptual framework several novel concepts and abstractions that are emerging in different areas without recognition of the basic commonalties, because of a lack of interaction and common terminology. This synthesis may suggest new applications for their broadly applicable MAS concepts.

Following this synthesis, Section 3 argues that integration of these concepts and abstractions in software modeling and design is not an incremental *evolution* of current models and methodologies, but a *revolution*, a radical change of paradigm [Kuh96] pioneered by the MAS community. This revolution will impact most research communities and will dramatically change how we conceive, model, and build "software components" and "software systems." Next generation software systems will no longer be modeled and designed as "mechanical" or "architectural" systems, but rather as "physical" or "intentional" systems. We try to identify the impact of such a change of paradigm in computer science and software engineering practices.

2 What's New?

The four characteristics identified in the introduction (situatedness, openness, local control, local interactions) affect most of today's software systems.

2.1 Situatedness

Today's computing systems are situated: they have an explicit notion of the environment in which components exist and execute, environmental characteristics affect their execution, and their components often explicitly interact with that environment.

Software systems have always been and will always be immersed in some sort of environment. For instance, the execution of a process in a multi-threaded machine is intrinsically affected by the dynamics of the multiprocessing system. Traditional modeling and engineering tries to mask the presence of the environment. In most the cases, specific objects and components "wrap" the environment and model it in terms of a "normal" component, so that the environment in itself does not exist as a primary abstraction. Unfortunately, the environment in which components live and with which they interact does exist and may impact execution and modeling:

- Several entities with which software components may need to interact are too complex in their structure and behavior to enable a trivial wrapping.
- For a system whose goal is to monitor (sense) and control (affect) a physical or logical (computational) environment, masking the environment is not natural. Instead, providing an explicit consciousness may be a primary application goal.

- The environment can have its own dynamics, independent of a software system's intrinsic dynamics. Wrapping the environment will introduce unpredictable non-determinism in the behavior of some parts of our applications.

For these reasons, both computer science and software engineering now tend to define a system's execution environment as a primary abstraction, explicitly defining both the "boundaries" separating the software system from its environment and the reciprocal influences of the two systems. This approach both avoids odd wrapping activities needed to model each component of the external world as an internal application entity, and allows a software system to deal more naturally with its activities in the real-world environment it is devoted to monitor and control. In addition, explicit modeling of the environment and its activities makes it possible to identify and confine clearly the sources of dynamics and unpredictability (and, thus, of non-formalizability [Weg97]), and concentrate on software components as deterministic entities that have to deal with a dynamic and possibly unpredictable environment.

Examples.

Control systems for physical domains (e.g., manufacturing, traffic control, home care, health care) are often built by explicitly taking into account unpredictable environmental dynamics via specific event-handling (or similar) policies [GusF01]. Similar problems arise in sensor and robot networks [IntGE00], where many components are spread in an unknown environment to explore and monitor it.

Mobile and pervasive computing recognizes the importance of context-awareness, e.g., applications' need to model environmental characteristics (e.g., those related to the sensed physical environment [HowM02] or to the locations of specific components [Pri01]) and environmental data (e.g., provided by an embedded infrastructure [ImiG00]) explicitly, rather than implicitly in terms of internal object attributes.

Applications intended to be immersed in the intrinsically dynamic Internet environment are typically engineered by defining the boundaries of the system in terms of "application," including the new application components to be developed, and "middleware," the environmental substrate in which components will be embedded [ZamJW01,CabLZ02]. Clearly identifying and defining such boundaries is a key point in web-application engineering. Several systems for workflow management and computer supported collaborative work are built around shared data space abstractions as the common execution environment for workflow processes and agents [Tol00].

Finally, several promising proposals in distributed problem solving and optimization (i.e., works on ant-based colonies [ParB01, ParBS02]), exploit 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 it intrinsically imply openness. The software system is no longer isolated, but becomes a permeable subsystem, whose boundaries permit reciprocal side-effects. This reciprocal influence between system and environment is often extreme and complex, making it difficult to identify boundaries clearly between the system and its environment.

In several cases, to achieve their objectives, software systems must interact with external software components, either to provide services and data, or to acquire them. More generally, different software systems, independently designed and modeled, are

likely to "live" in the same environment and explicitly interact with each other. These 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 systems may come to life and die in an environment independently of each other, or when a software system (or its components) can explicitly move across different environments during its life. These characteristics introduce additional problems.
- When component in different software systems interact, and when components move across different environment, it may become hard if not impossible to identify clearly the system to which a component belongs. In other words, it becomes difficult to understand clearly the boundaries of software systems.
- When a component comes to life in an environment (or is somehow moved to a specific environment), it must somehow be made aware of what is around in the environment, what other components are there for interaction, and how it can interact with newly entering systems safely for both the systems and itself.
- Enabling 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 these problems, computer science and software engineering are starting to consider the problem of not only modeling the environment in which systems execute, but also of understanding and modeling the dynamic changes in system structure. Also, the need to preserve coherency of the system despite openness may require identifying and enacting specific policies, intended to support the re-organization of the software system in response to changes in its structure, or to enact contextual laws on the activity of those components entering the system. The general aim is to increase the ability to control system execution despite its dynamic structural changes.

Examples.

Control systems for critical environments, such as traffic control systems, public telephony services, health care systems, manufacturing systems, and sensor-based monitoring, run continuously, cannot be stopped, and sometimes cannot even be removed from the environment in which they are embedded. Nevertheless, these systems need to be updated, and their environment is likely to change frequently, with the addition of new physical components and, consequently, of new software components and software systems [Ten00]. For all these systems, managing openness and a system's ability to re-organize itself automatically (e.g., by establishing new effective interactions with new components and/or by changing the structure of the interaction graph [IntGE00, RipIF02]) is of dramatic importance, as is the ability of a component to enter new execution contexts safely and effectively.

Mobility, whether of users, software, or devices, moves the concept of openness to the extreme, by making components actually move from one context to another during their execution, changing the structure of the software system executing in that context [Whi97, CabLZ02]. This requires not only the ability of components to learn about their new context, but also the ability to organize and control component interactions in the context [PicMR00]. Such concepts are exacerbated in mobile ad-hoc network-

ing, where interactions must be made fruitful and somehow controllable despite the lack of any intrinsic structure and the dynamics of connectivity [Bro98]

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 also be able to enact security and resource control policy in their local context, e.g., a given administrative domain [CabLZ02]. E-marketplaces are the most typical examples of this class of open Internet applications [NorST98]. P2P systems stands to Internet applications as ad-hoc networking stands in mobile computing system: components must be allowed to fruitfully and properly interact without any pre-determined interaction structure and by surviving the dynamics of services and data availability [RowD01, RipIF02].

2.3 Local Control

The "flow of control" concept has always been one of the key aspects 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 becomes meaningless.

Independent software systems have their own autonomous flows of control, and their mutual interactions do not imply any join of these 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 makes the concept of "global execution control" disappear. This trend is exacerbated by the fact that not only do 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 ones (e.g., "normal" objects, functions, etc.).

Having multiple threads of control in a system is not novel, and concurrent and parallel programming are well-established paradigms. However, the main motives for multiple threads in traditional concurrent and parallel programming are efficiency and performance. Most approaches try to limit the independence of these multiple threads as much as possible, via strict synchronization and coordination mechanisms, to preserve determinism and high-level control over applications. Today's autonomy of application components has different motives and has to be handled differently.

- In an open world, autonomy of execution enables a component to move 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 whose evolution must be monitored and controlled, an autonomous component can be effectively delegated to take 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 also integrate computer-based systems, which are by their very nature autonomous systems, and can be modeled accordingly.
- As the size of a software system increases, both performance and conceptual simplicity require delegating control to components. Coordinating a global flow of

control among a very large number of components may become unfeasible. Autonomy then becomes an additional dimension of modularity [Par97].

Our concept of autonomy refers not only to those software components that are designed as autonomous, but also to those that can be perceived as autonomous.

Examples.

Almost all modern software systems integrate autonomous components. Weak autonomy is the ability of a component to react to and handle events (e.g., graphical interfaces or embedded sensors). Strong autonomy implies that a component integrates an autonomous thread of execution, and can execute proactively. In most modern control systems, control is not simply reactive but proactive, realized via a set of cooperative autonomous processes or, often, via embedded complete computer-based systems interacting with each other [GusF01] or distributed sensor nets [IntGE00].

The integration in complex distributed applications and systems of (software running on) mobile devices of any type can be tackled only by modeling them in terms of autonomous software components [PicMR00].

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 and by decentralized management rather than by the actual need of autonomous concurrent activities.

2.4 Local Interactions

Directly deriving from these 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 a specific environment, execute in the context of a specific (sub)system, and are delegated to perform some task in autonomy. Taken all together, these aspects naturally lead to a strict enforcement of locality in interactions. In fact:

- Autonomous components can interact with the environment in which they are immersed, by sensing and affecting it. If the environment is physical, the amount of the physical world a single component can sense and affect is locally bounded by physical laws. If the environment is logical, minimization of conceptual and management complexity still favor modeling it in local terms 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 to which they belong, that is, locally to their system. In open work, however, a component of a system can also interact with (components of) other systems. In these cases, minimization of conceptual complexity suggests modeling the component in terms of a component that has temporarily "moved" to another context, and that interacts locally in the new context [CabLZ02].
- In an open world, components need some form of context-awareness to execute and interact effectively. For a component to be made aware of its context effectively (and efficiently), this context must necessarily be locally confined.

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, tracking and controlling concurrent, autonomous, and autonomously initiated interac-

tions is much more difficult than in object-based and component-based applications, even if these interactions are strictly local.

Examples.

Control and manufacturing systems tend naturally to enforce local interactions. Each control component is delegated control of a portion of the environment, and its interactions with other components are usually limited to those that control neighboring portions of that environment, with which it typically is strictly coordinated.

In mobile computing, including applications for wearable systems and sensor networks, the very nature of wireless connections forces locality in interactions. Since wireless communication has limited range, a mobile computing component can directly interact – at a given time – only with a limited portion of the world [PicMR00].

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 [CabLZ02, Whi97].

2.5 A General Model

In sum, software systems increasingly follow the scheme of Figure 1. Software systems (dashed ellipses) are made up of autonomous components (black circles), interacting locally with each other and possibly with components of other systems. Some components may belong to several systems at different times. Systems and components are immersed in an environment, typically modeled as a set of environment partitions. Components in a system can sense and affect a local portion of the environment. Also, since the portions of the environment that two components may access may overlap, two components may interact indirectly via the environment.

The scenario of Figure 1 is very different from component-based and object-based programming, and matches the prevailing model of agent-based computing [Jen01]. Agents are situated software entities (they live in an environment) that execute autonomously (have local control of their actions) and interact with other agents. Local interactions are often promoted, although they may not be explicitly mentioned as part of the model. Despite this fact, most of the scientists working on agent-based computing still focus mostly on the AI aspects of the discipline, without noticing (or simply deliberately ignoring) that agents and agent-based computing have the potential to be a general model for today's computing systems and that different research communities face similar problems and are starting adopting similar modeling techniques. Similarly, outside the AI community, computer scientists often fail to recognize that they are already doing systems that can be assimilated to and modeled as agent-based systems.

The issues discussed

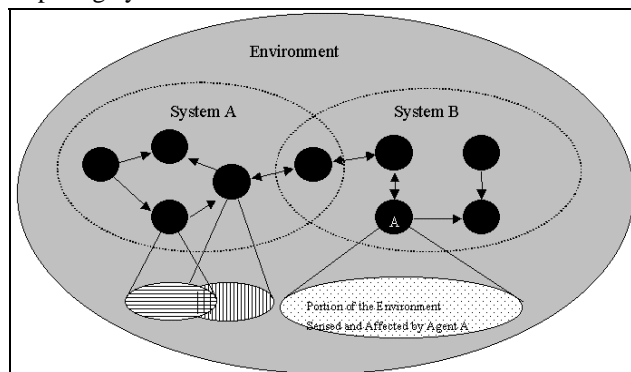


Fig. 1. The Scenario of Modern Software Systems

in this paper can help clarify these relationships and the need for more interaction between different research communities, due to the strong similarities (we are tempted to say isomorphism) between the addressed problems.

3 Changing our Attitudes

Traditionally, software systems are modeled from a mechanical stance, and engineered from a design stance. Computer scientists are both burdened and fascinated by the urge to define suitable formal theories of computation, to prove properties of software systems, and to provide a formal framework for engineering. Software engineers are used to analyzing the functionality that a system should exhibit in a top-down way, and to designing software architectures as reliable multi-component machines, capable of providing the required functionality efficiently and predictably.

In the future, scientists and engineers will have to design software systems to execute in a world where innumerable autonomous, embedded, and mobile software components are already executing and locally interacting with each other and with the environment. Such a scenario may require untraditional models and methodologies.

3.1 How Will Computer Science Change?

Modeling and handling systems with many components can be feasible if 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 conceptually and computationally infeasible, to track them one by one and, so, to describe precisely the behavior of a system in terms of the behavior of its components. In addition, as several software systems are distributed and subject to decentralized control, or possibly embedded in some unreachable environment [IntGE00], tracking components and controlling their behavior is simply impossible. Such systems can only be described and modeled as a whole, in terms of macro-level observable characteristics, just as a chemist describes a gas in terms of macro properties like pressure and temperature.

This 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 interactions. One may argue that since interactions tend to local (Subsection 2.4), this should not represent a big problem and, instead, it should be quite easy to control the effect of interactions. Unfortunately, the effect of local interactions can propagate globally and be very difficult to predict [PriS91]. Even if one knows the initial status of the system very accurately, nonlinearities can amplify small deviations to become arbitrarily great.

As an additional problem, software systems will execute in an open and dynamic scenario, where new components can be added and removed at any time, and where some of these components can autonomously move from one part of the system to another. Thus, it is difficult to predict and control precisely not only the global dynamic behavior of the system, but also how such behavior can be influenced by such openness. In fact, it has been shown that the effects of interactions of autonomous active components strongly depends on the structure of the interaction graph [Wat99]: strictly local interactions can produce a dynamic behavior that is completely different from the one emerging as soon as some form of non-locality in interactions is introduced, changing the structure of the interaction graph. Such problems can emerge in

open systems, where the possibility for components to join and leave a system at any time dynamically changes the interaction graph and, possibly, its dynamic behavior.

Situatedness causes a similar problem. Modern thermodynamic and social sciences show that environmental forces can produce strange large scale dynamic behaviors in situated physical, biological, and social systems [PriS91]. We can expect similar effects in situated software systems, because of the dynamics of the environment.

These problems will force computer scientists to change their attitudes dramatically in modeling complex software systems. Concurrency and interactivity already challenge the dream of dealing with traditionally formalizable systems [Weg97], and it will become even more impractical in the future. Traditional formalisms can deal effectively with only small portions of large systems. The next challenge is to find alternative models or, more radically, to adopt a new scientific background for the study of software systems, enabling us to study, predict, and control the properties of a system and its dynamic behavior despite the inability to control its individual components.

Signals. Some signals of this trend are appearing in the research community.

Recent study and monitoring activities on the Web [CroB96, AlbJB99] and on P2P networks [RipIF02] show that unpredictable and large-scale behaviors are already here, requiring new models and tools for their description. For instance, traditional Web caching policies are no longer effective when peculiar dynamic Web-access patterns emerge [CroB96] and traditional reliability models fall short due to the specific emergent characteristics of Web and P2P networks [AlbJB00, RipIF02].

Some approaches to model and describe software systems in terms of thermodynamic systems have already been proposed [ParB01, ParBS02]. The ideas behind such research are twofold: to provide synthetic indicators capable of measuring how closely the system is approaching the desired behavior, and to provide tools to measure the influence of the environmental dynamics on the system. To some extent, a similar approach has been adopted in the area of massively parallel computing, where the need to measure specific global systems properties dynamically requires the introduction of synthetic indicators [CorLZ99]. Similarly, it has been recognized that modeling large and dynamic (overlay) networks can only be faced by introducing macro properties rather than by direct representation of the network [AlbJB99, RipIF02].

One area that clearly shows such a trend 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 (matching the characteristics of today's software systems, Subsection 2.5) have shifted the emphasis to the concept of "intentional" intelligence, i.e., the capability of a component or of a system to behave autonomously so as to achieve a given goal. Organizational [ZamJW01] and social science [MosT95] are starting to influence research, as it is recognized that the behavior of a large scale software system can be assimilated more appropriately 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 members, than to a logical or mechanical system. Similarly, inspiration for computer scientists comes increasingly from the complex mechanisms of biological ecosystems, as well as the mindsets of biological sciences [HubH93, GusF01].

We expect theories and models from complex dynamical systems, modern thermodynamics, biology, social science, and organizational science, will have to become more and more the sine-qua-non cultural background of the computer scientist.

3.2 How Will Software Engineering Change?

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

Today, software systems are designed to exhibit specific, predictable, and deterministic behavior at any level, from single units up to the whole system. However, in the presence of autonomous components situated in an open and dynamic environment, obtaining predictable behavior of a multi-component system in mechanical and deterministic terms is not feasible. The next challenge for the effective construction 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 guarantee that the system as a whole will behave reasonably close to an ideal desired behavior despite the lack of exact knowledge about its micro-behavior. For instance, by adopting a "physical" attitude toward software design, a possible approach could be to build a system that, despite uncertainty on the initial conditions, is able to reach a given stable basin of attraction. By adopting a "teleological" attitude, one could 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.

The design of a system must also explicitly take into account the fact that the system will be immersed in an open and dynamic environment, and that the behavior of the system cannot be designed in isolation. Rather, the environment and its dynamics, as well as those software components that can enter and leave the systems, must become primary components of the design abstractions. A *defensive* design treats these factors as sources of uncertainty that can somehow be damaging to the global behavior of a system and that the system should be prepared to face. An *offensive* design considers openness and environmental dynamics as additional design dimensions to be exploited with the possibility of improving the behavior of the system [ParBS02].

Signals. Again, a few exemplars adopt this novel 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 a global goal via local actions and local interactions, despite the dynamics of the environment [Cyb89]. The design of these policies is, in several cases, inspired by physical phenomena such as diffusion, which can re-establish global equilibrium in dynamic situations [CorLZ99], despite the fact that such equilibrium will never be perfect but always locally perturbed.

Systems of ant colonies designed bottom-up can solve complex problems that resist traditional approaches, using very simple autonomous components interacting via a dynamic environment [Par97]. The idea is to mimic in software the behavior of insect colonies living in a dynamic world and able to solve, as a group and via the work sacrifice of a large number of worker insects (which translates in computational sacrifice to be paid), problems that have an interesting computational counterpart (i.e., sorting and routing). In that case, the environmental dynamics plays a primary role in design and in the emergence of specific useful behaviors.

Social phenomena like epidemics have inspired distributed information in dynamic networks of components, such as mobile ad-hoc networks [MitV00], despite the inability of exactly controlling the information paths and structure of a network. The dynamics of the network is both a source of uncertainty and a useful property to guarantee that a message propagates to the whole network in a reasonable time. P2P information dissemination systems similarly exploit the dynamic properties of a spontaneously generated community network [PielF02].

Both the dimension and the intrinsic dynamics of the network make it impossible to detect the structure of a network and of its components, challenging the whole concepts of information routing and information retrieval. In different areas (e.g., pervasive and mobile computing [Adj99], P2P Internet computing [RowD01, Sto01], middleware [CarRW01, MamLZ02], sensor networks [IntGE00]), there is a general understanding that the dynamics of networks require novel approaches to information retrieval and routing focused on content rather than paths. In other words, the basic idea is that the path to information sources/destinations should be dynamically found by relying on abstract overlay networks, dynamically and automatically reshaping themselves in response to system dynamics. Data or queries follow the shape of the overlay network just as a ball rolls down a surface. Whatever the final destination, the paths that emerge always proceed in a “good” direction and eventually reach a point where the nodes in the network and the routed message are mutually appropriate.

Despite its intrinsic uncertainty, biological evolution is also likely to be a useful tool for software engineers. For instance, though cellular automata can perform complex computations, it is almost impossible to understand which rules must be imposed on cells and on their interaction to produce a system with required properties. An approach that has been successfully experienced is to make cellular automata rules evolve (e.g., via genetic algorithms) and eventually come up with specific rules leading to the desired global behavior of the automata, without anyone having directly designed such behaviors [Sip99].

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

For software systems conceived mechanically, testing mainly amounts to analyzing system state transitions to see if they correspond to those designed or if some of the components exhibit bad state transitions, i.e., bugs. Such work is very hard for large (even non-concurrent) systems, and may become impossible when autonomy and environmental dynamics produce a practically uncountable number of states. Thus, the testing criterion for a large software system will no longer be the absence of errors, but rather its ability to behave as needed as a whole, independently of the exact behavior of its components and of their initial conditions [Huh01]. In an existing dynamic environment, where other systems are already executing and cannot be stopped, software cannot be simply tested and evaluated in terms of its capability to achieve the required goal. The test must also evaluate the effect of the environment on the software system and the effects of the software system on the environment. The better and more robust a system, the more reliably it can advance its goals independently of the dynamics of the environment, and the lower its impact on the surrounding environment and, thus, on other software systems. As a notable example, several approaches to content-based routing tend to evaluate their proposals in terms of how much the system can tolerate

network dynamics by still exhibiting reasonably good behaviors, how fast the overlay network can re-organize in response to dynamic changes, and how the addition and removal of components impact network behavior [AlbJB00].

Software maintenance will change too. Autonomy and openness mean that no software system will be stable, but to some extent needs maintenance continually due to changed conditions. When a large software system no longer behaves as needed (e.g., when external conditions require changes), update will no longer imply stopping the system, rebuilding it, and retesting it. Instead, it will imply intervening externally, by adding new components with different attitudes and by removing some of its existing components so as to change, as needed, the overall behavior of the system. The openness and situatedness of the system and the autonomy of its components can also make maintenance and updating smoother and less expensive, in that the system is already designed to tolerate and support dynamic changes in its structure.

3.3 Discussion

Humans increasingly rely on software systems for everyday activities, as well as for control of critical situations. One possible criticism of this approach is that such applications cannot be engineered in the way we have envisioned. Instead, one could insist that a software system exhibit a predictable and fully controllable behavior in each of its parts, and that the duty of a software engineer is to produce such reliable systems.

A few MAS researchers are pioneering such new perspective. But many others, aware that emergent behavior is likely to appear in open systems of autonomous components, consider all such behavior as undesirable. For instance, pessimists foresee the death of P2P approaches because of their unreliability and the impossibility of proper modeling. In our opinion, this is not the correct approach. Constraining the behavior of a highly interactive system may sacrifice most of its computational power and may require much more waste of resources to obtain the same functionalities. The engineering work needed to make a system fully controllable may be greater than the work needed to produce emergent behavior that is useful, reliable, and stable. Finally, constraining "by design" a software system may simply make it lose the properties needed to support openness and to tolerate environmental dynamics.

In any case, we are aware that, for our vision on software engineering to become practical and used, much theoretical, methodological, and experimental work is required, paving the way for a suitable set of conceptual tools and frameworks to be exploited by the engineers of next generation software systems.

4 Conclusions

Modern software systems, in different application areas, exhibit characteristics that make them very different from the software systems to which we, as scientists and engineers, are accustomed. These characteristics are likely to impact dramatically 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 from our traditional "design" attitude, implying a mechanical perspective, to an "intentional" attitude, requiring physical, biological, and teleological perspectives. Despite the opposing forces and the difficulties inherent in any revolutionary phase, including the need of restructuring our cultural background, this revolu-

tion will open the door to new interesting research and engineering challenges in which, hopefully, the multi-agent research community will hold a leading position.

References

- [Abe00] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman and R. Weiss, "Amorphous Computing", *Communications of the ACM*, 43(5), May 2000.
- [AboM00] G. D. Abowd, E. D. Mynatt, "Charting Past, Present and Future Research in Ubiquitous Computing", *ACM Transactions on Computer-Human Interaction*, 7(1):29-58, March 2000.
- [Adj99] W. Adjie-Winoto, E. Schwartz, H. Balakrishna, J. Lilley, "The Design and Implementation of an Intentional Naming Systems", 17th ACM Symposium on Operating Systems Principles (SOSP '99), ACM, 1999.
- [AlbJB99] R. Albert, H. Jeong, A. Barabasi, "Diameter of the World Wide Web", *Nature*, 401:130-131, 9 Sept. 1999.
- [AlbJB00] R. Albert, H. Jeong, A. Barabasi, "Error and Attack Tolerance of Complex Networks", *Nature*, 406:378-382, 27 July 2000.
- [Bro98] J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu, J. Jetcheva, "A Performance Comparison of Multi-hop Wireless Ad-Hoc Network Routing Protocols", 3rd ACM/IEEE Conference on Mobile Computing and Networking, October 1998.
- [Bus00] S. Bussmann, "Self-Organizing Manufacturing Control: an Industrial Application of Agent-Technology", 4th IEEE International Conference on Multiagent Systems, Boston (MA), July 2000, pp. 87-94.
- [CabLZ02] G. Cabri, L. Leonardi, F. Zambonelli, "Engineering Mobile Agent Applications via Context-Dependent Coordination", *IEEE Transactions on Software Engineering*, 2002, to appear.
- [Cap97] F. Capra, *The Web of Life: The New Understanding of Living Systems*, Doubleday, Oct. 1997.
- [CarRW01] A. Carzaniga, D. S. Rosenblum, A. L. Wolf, "Design and Evaluation of a Wide-Area Event-Notification Service", *ACM Transactions on Computer Systems*, 19(3):332-383, Aug. 2001.
- [CorLZ99] A. Corradi, L. Leonardi, F. Zambonelli, "Diffusive Load Balancing Policies for Dynamic Applications", *IEEE Concurrency*, 7(1):22-31, 1999.
- [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.
- [Est02] D. Estrin et al., "Connecting the Physical World with Pervasive Networks", *IEEE Pervasive Computing*, 1(1):59-69, January 2002.
- [GusF01] R. Gustavsson, M. Fredriksson, "Coordination and Control in Computational Ecosystems: A Vision of the Future, in *Coordination of Internet Agents*, A. Omicini et. al (Eds.), Springer Verlag, pp. 443-469, 2001.
- [HowM02] A. Howard, M. J. Mataric, "Cover Me! A Self-Deployment Algorithm for Mobile Sensor Networks", *International Conference on Robotics and Automation*, 2002, to appear.
- [HubH93] B. A. Huberman, T. Hogg, "The Emergence of Computational Ecologies", in *Lectures in Complex Systems*, Addison-Wesley, 1993.
- [Hub01] M. Huhns, "Interaction-Oriented Programming", 1st International Workshop on Agent-Oriented Software Engineering, LNCS No. 1957, Jan. 2001.
- [IntGE00] C. Intanagonwiway, R. Govindam, D. Estrin, "Directed Diffusion: a Scalable and Robust Communication Paradigm for Sensor Networks", 5th ACM/IEEE Conference on Mobile Computing and Networking, Boston (MA), Aug. 2000, pp. 56-67.

- [Jen01] N. R. Jennings, "An Agent-Based Approach for Building Complex Software Systems", *Communications of the ACM*, 44(4):35:41, 2001.
- [Kuh96] T. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, 3rd Edition, Nov. 1996.
- [MamLZ02] M. Mamei, L. Leonardi, F. Zambonelli, "A Physically Grounded Approach to Coordinate Movements in a Team", 1st International Workshop on Mobile Teamwork at ICDCS, IEEE CS Press, July 2002.
- [MitV00] W. G. Mitchener, A. Vahdat, "Epidemic Routing for Partially Connected Ad-Hoc Networks", Duke Technical Report, No. CS-2000-06, July 2000.
- [MosT95] Y. Moses, M. Tenneholtz, "Artificial Social Systems", *Computers and Artificial Intelligence*, 14(3):533-562, 1995.
- [NorSR98] P. Noriega, C. Sierra, J. A. Rodriguez, "The Fishmarket Project. Reflections on Agent-mediated institutions for trustworthy E-Commerce", 1st Workshop on Agent Mediated Electronic Commerce (AMEC-98), 1998.
- [Par97] V. Parunak, "Go to the Ant: Engineering Principles from Natural Agent Systems", *Annals of Operations Research*, 75:69-101, 1997.
- [ParB01] V. Parunak, S. Bruekner, "Entropy and Self-Organization in Agent Systems", 5th International Conference on Autonomous Agents, ACM Press, May 2001.
- [ParBS02] V. Parunak, S. Bruekner, J. Sauter, "ERIM's Approach to Fine-Grained Agents", NASA/JPL Workshop on Radical Agent Concepts, Greenbelt (MD), Jan. 2002.
- [PicMR00] G. P. Picco, A.M. Murphy, G.-C. Roman, "Software Engineering for Mobility: A Roadmap", in *The Future of Software Engineering*, A. Finkelstein (Ed.), ACM Press, pp. 241-258, 2000.
- [Pri01] N.B. Priyantha, A.K.L. Miu, H. Balakrishnan, S. Teller, "The Cricket Compass for Context-aware Mobile Applications", 6th ACM/IEEE Conference on Mobile Computing and Networking, Rome (I), July, 2001.
- [PriS91] I. Prigogine, I. Steingers, *The End of Certainty: Time, Chaos, and the New Laws of Nature*, Free Press, 1997.
- [RipIF02] M. Ripeani, A. Iamnitchi, I. Foster, "Mapping the Gnutella Network", *IEEE Internet Computing*, 6(1):50-57, Jan.-Feb. 2002.
- [RowD01] A. Rowstron, P. Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems", 18th IFIP/ACM Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, Nov. 2001.
- [Sip99] M. Sipper. "The Emergence of Cellular Computing", *IEEE Computer*, 37(7):18-26, July 1999.
- [Sto01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications", ACM SIGCOMM Conference 2001, San Diego (CA), Aug. 2001.
- [Tol00] R. Tolksdorf, "Coordinating Work on the Web with Workspaces", 9th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, Gaithersburg (MA), IEEE CS Press, June 2000.
- [Ten00] D. Tennenhouse, "Proactive Computing", *Communications of the ACM*, 43(5):43-50, May 2000.
- [Wat99] D. Watts, *Small Worlds : The Dynamics of Networks between Order and Randomness*, Princeton University Press (Princeton, NJ), 1999.
- [Weg97] P. Wegner. "Why Interaction is More Powerful than Algorithms", *Communications of the ACM*, 1997.
- [Wei93] M. Weiser, "Hot Topics: Ubiquitous Computing", *IEEE Computer*, 26(10), October 1993.

- [Whi97] J. White, "Mobile Agents", in *Software Agents*, J. Bradshaw (Ed.), AAAI Press, Menlo Park (CA), pp. 437-472, 1997.
- [ZamJW01] F. Zambonelli, N. R. Jennings, M. J. Wooldridge, "Organizational Rules as an Abstractions for the Analysis and Design of Multi-agent Systems, *International Journal of Knowledge and Software Engineering*, 11(4), April. 2001.
- [MitV00] W. G. Mitchener, A. Vahdat, "Epidemic Routing for Partially Connected Ad-Hoc Networks", Duke Technical Report, No. CS-2000-06, July 2000.