The Knowledge Engineering Review, Vol. 18:4, 329–342. © 2004, Cambridge University Press
DOI: 10.1017/S0269888904000104 Printed in the United Kingdom

Towards a paradigm change in computer science and software engineering: a synthesis

FRANCO ZAMBONELLI[1] and H. VAN DYKE PARUNAK[2]

[1]*Dipartimento di Scienze e Metodi dell'Ingegneria, Universitá di Modena e Reggio Emilia, Via Allegri 13, 42100 Reggio Emilia, Italy*
E-mail: franco.zambonelli@unimo.it
[2]*Altarum Institute, 3520 Green Ct, Suite 300, Ann Arbor, MI 48105, USA*
E-mail: van.parunak@altarum.org

Abstract

In this paper, we identify and analyse a set of characteristics that increasingly distinguish today's complex software systems from ''traditional'' ones. Several examples in different areas show that these characteristics are not limited to a few application domains but are widespread. Then, we discuss how these characteristics are likely to impact dramatically on the very way software systems are modelled and engineered. In particular, we appear to be on the edge of a radical shift of paradigm, which is about to change our very attitudes in software systems modelling 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 (Tennenhouse, 2000). Computer systems will be embedded in every object (Estrin *et al*., 2002; Gellersen *et al*., 2002), e.g. in our physical environments, our clothes and furniture, and even our bodies. Wireless technologies will make network connectivity pervasive (Abowd & Mynatt, 2000), and every computing device will be connected in some network, whether the ''traditional'' Internet or *ad hoc* local networks (Broch *et al*., 1998). Finally, computing systems will always be active so as to perform some activity on our behalf, e.g. to improve comfort at home, or to control and automate manufacturing processes (Nagbal *et al*., 2003; Bussmann, 2000).

This scenario does not simply affect the design and development of software systems *quantitatively*, in terms of the number of components and effort required. There will also 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;

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

- *openness*: software systems are subject to decentralised management and can dynamically change their structure;
- *locality in control*: the components of software systems represent autonomous and pro-active loci of control;
- *locality in interactions*: despite living in a fully connected world, software components interact with each other according to local (geographical or logical) patterns.

The above four characteristics are those that are typically used to characterise multi-agent systems in the research community of distributed artificial intelligence (Jennings, 2001). However, if we go into the details of the above characteristics, we can see that several research communities focused on different topics (e.g., manufacturing and environmental control systems (Bussmann, 2000; Estrin *et al*., 2002; Intanagonwiway *et al*., 2000), mobile and pervasive computing (Weiser, 1993; Abowd & Mynatt, 2000), the Internet (Cabri *et al*., 2002) and P2P computing (Ripeani *et al*., 2002)) are already recognising their importance, and are already adapting their models and technologies to take them into account. Thus, our first contribution is to attempt to synthesise in a single conceptual framework several novel concepts and abstractions that are emerging in different areas without recognition of the basic commonalities, because of a lack of interaction and common terminology.

Following this synthesis, we argue that the integration of these concepts and abstractions in software modelling and design does not simply represent a proper evolution of current models and methodologies. Instead, we anticipate a real revolution or (using Kuhn's term (Kuhn, 1996)) a scientific change of paradigm, likely to impact most research communities horizontally and to change the very way we conceive, model, and build ''software components'' and ''software systems''. Clear signals testify that next-generation software systems will no longer be modelled and designed in terms of ''mechanical'' or ''architectural'' systems, but rather in terms of ''physical'' or ''intentional'' systems.

2 What's new?

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

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, they are affected in their execution by environmental characteristics, and their components often explicitly interact with that environment.

We emphasise that software systems have never worked in isolation, but have always been and always will be immersed in some sort of environment. For instance, the execution of a running process in a multi-threaded machine is intrinsically affected by the dynamics of the multiprocessing system. However, traditional modelling and engineering approaches have always tried to mask the presence of the environment. In most cases, specific objects and components ''wrap'' the environment to 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 indeed exist and may impact on application execution and on application modelling.

- Several entities with which software components may need to interact are too complex in their structure and behaviour 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 (computational) environment, masking such an environment is not a

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

natural choice. Instead, providing an explicit consciousness may become a primary application goal.
- The environment can have its own dynamics, independent of a software system's intrinsic dynamics. Wrapping the environment in an application entity will introduce forms of unpredictable non-determinism inside applications.

For the above reasons, the current trend in both computer science and software engineering is to define the environment in which a software system executes explicitly as a primary abstraction, explicitly defining both the ''boundaries'' separating the software systems and its environment, and the reciprocal influences of the two systems (Odell *et al*., 2003). 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 in a more natural way with the real-world environment that the software system itself may be devoted to monitor and control. In addition, explicit modelling of the environment and of its activities makes it possible to identify and confine clearly the sources of dynamics and unpredictability (and, thus, of non-formalisability (Wegner, 1997)), and concentrate on our software components in terms of deterministic entities that have to deal with a dynamic and possibly unpredictable environment.

2.1.1 Examples

Control systems for physical domains (e.g., manufacturing, traffic control, home care, and health care) tend 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 (or the like) (Gustavsson & Fredriksson, 2001). Similar problems arise in sensor and robot networks (Estrin *et al*., 2002), where a multitude of components are spread throughout an unknown physical environment with the duty of exploring and monitoring it.

Mobile and pervasive computing applications recognise the primary importance of context-awareness as the need for applications to maintain explicit models of environmental characteristics (e.g., characteristics related to the sensed physical environment (Howard *et al*., 2002) or to the position of specific components in an environment (Priyantha *et al*., 2001; Nagpal *et al*., 2003)) and environmental data (e.g., data provided by some embedded infrastructure in the environment, rather than implicit models (in terms of the internal attributes of objects).

Internet applications and Web-based systems, which must be immersed in the existing and intrinsically dynamic Internet environment, are typically engineered by clearly 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) (Zambonelli *et al*., 2003; Cabri *et al*., 2002). Clearly identifying and defining such boundaries is one of the key points in Web-application engineering. Similarly, several systems for workflow management and computer-supported collaborative work are built around shared data space abstractions, to be exploited as the common environment for the execution of workflow processes and agents (Tolksdorf, 2000; Ricci *et al*., 2003).

As a final example, it is worth noting that several promising proposals in the area of distributed problem solving and optimisation (i.e. work on ant-based colonies (Parunak & Bruekner, 2001; Parunak *et al*., 2002)) are based on a model cantered 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 it intrinsically imply some sort of openness of a software system. In fact, the system can no longer be conceived as an isolated world, but must instead be considered as a permeable sub-system, whose boundaries permit reciprocal side-effects.

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

In several cases, to achieve their objectives, software systems must interact with external software components, either to provide them with services or data, or to acquire them. More generally, different software systems, independently designed and modelled, 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, and simply enabling interoperability is not enough when software systems may come to life and die in an environment independently of each other, in a very dynamic way, or when a software system (or its components) can explicitly move across different environments during its life. These characteristics introduce additional problems.

- When a component interacts with other components in other software systems, and when components move across different environments, it may become difficult if not impossible to clearly identify the system to which a component belongs. In other words, it becomes difficult to understand the boundaries of software systems clearly.
- When a component comes to life in an environment (or is moved to a specific environment) it must somehow be made aware of its environmental context, what other components there are eligible for interaction, and how it can interact in the newly entered system without endangering either the system or 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 behaviour of these software systems and even of a single software system.

Due to the above problems, software development may require the problem of not only modelling the environment in which systems execute, but also of understanding and modelling dynamic changes in the system structure to be faced. Also, the need to preserve the coherence of the system despite openness may require the identification and enactment of specific policies, intended to support the re-organisation of the software system in response to the changes in its structure, or the enactment of contextual laws on the activity of those components entering the system. The general aim is to increase the ability to control the system execution despite its dynamic structural changes.

2.2.1 Examples

Control systems for critical physical domains typically run forever, cannot be stopped, and sometimes cannot even be removed from the environment in which they are embedded. Nevertheless, these systems need to be continuously updated, and the environment in which they live is likely to change frequently, with the addition of new physical components and, consequently, of new software components and software systems (Tennenhouse, 2000). For all these systems, managing openness and the capability of a system to re-organise itself automatically (e.g., by establishing new effective interactions with new components and/or by changing the structure of the interaction graph (Intanagonwiway *et al.*, 2000; Ripeani *et al.*, 2002)) is very important, as is the ability of a component to enter new execution contexts safety and effectively. Also, with respect to pervasive computing systems (e.g., sensor networks and networks of embedded computer-based systems), lack of power or simply communication unreachability (Estrin *et al.*, 2002) can make nodes come and go unpredictably, thus requiring frequent restructuring of communication patterns.

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, thus changing the structure of the software system executing in that context (White, 1997; Cabri *et al.*, 2002). This requires not only the capability of components to acquire knowledge about the newly entered context in which they execute, but also the capability to organise and control component interactions in the context (Picco *et al.*, 2000). Such challenges are exacerbated in the context of mobile *ad hoc* networking, where interactions must be fruitful and somehow controllable despite the lack of any intrinsic structure and the dynamics of connectivity (Broch *et al.*, 1998).

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

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 policies in their local context, e.g. a given administrative domain (Cabri *et al*., 2002). E-marketplaces are the most typical examples of this class of open Internet applications (Rodriguez-Aguilar & Sierra, 2002). P2P systems (e.g., Gnutella and Freenet) correspond to Internet applications as *ad hoc* networking corresponds to mobile computing systems. In these cases, components must be allowed to interact fruitfully and properly without any pre-determined interaction structure and by surviving the dynamics of services and data availability (Rowstron & Drischel, 2001; Ripeani *et al*., 2002).

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 modelling and design of open software not only makes the concept of ''software system'' rather vague, as discussed in Section 2.2, but it also makes the concept of ''global execution control'' disappear. This trend is exacerbated by the fact that each independent system not only has its own flow of control, but also may have components that are individually autonomous. 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.).

We are aware that concurrent and parallel programming are well-established paradigms, and that applications with multiple threads and processes have been around for a long time. However, traditional concurrent and parallel programming, in order to improve efficiency and performance, exploit multiple flows of execution by avoiding making them multiple threads of control. In fact, most traditional approaches limit as much as possible the autonomy of these multiple flows of execution, via strict synchronisation and coordination mechanisms, with the goal of preserving determinism and control in application execution. This tendency can be seen even in some approaches to multi-agent systems engineering, aimed at somehow limiting agents' autonomy (Wooldridge & Jennings, 1998). However, today's autonomy of application components have different motivations and must 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 acknowledgment 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 modelled accordingly;
- as the size of a software system increases, the need to delegate control to components is no longer simply a matter of performance, but becomes a matter of conceptual simplicity; in fact, coordinating a global flow of control among a large number of components may become unfeasible, and autonomy can become an additional dimension of modularity (Parunak, 1997).

2.3.1 Examples

Almost all of today's software systems integrate autonomous components. At its weakest, autonomy reduces to the ability of a component to react to and handle events, as can be the case

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

for 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 pro-active way.

This is the case in most modern control systems for physical domains, in which control is not simply reactive but pro-active, realised via a set of cooperative autonomous processes or, as is often the case, via embedded complete computer-based systems interacting with each other (Gustavsson & Fredriksson, 2001) or via distributed sensor networks (Intanagonwiway *et al*., 2000; Estrin *et al*., 2002; Nagpal *et al*., 2003).

The integration in complex distributed applications and systems of (software running on) mobile devices of any type can be tackled only by modelling them in terms of autonomous software components (Picco *et al*., 2000; Cabri *et al*., 2002).

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 decentralised management rather than by the actual need for autonomous concurrent activities.

2.4 Local interactions

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

By now, we have delineated a scenario in which software system components are immersed in a specific environment, execute in the context of a specific (sub)system, and are delegated to perform some task autonomously. Taken all together, these aspects naturally lead to a strict enforcement of locality in interactions. In fact, we can consider the following.

- 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, minimisation of conceptual and management complexity still favours modelling 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, minimisation of conceptual complexity suggests modelling the component as though it has temporarily ''moved'' to another context, and in which it interacts locally (Cabri *et al*., 2002).
- In an open world, components need some form of context-awareness to execute and interact effectively. However, 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 scale of distribution increases. 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.

2.4.1 Examples

Control systems for physical domains tend to enforce local interactions by 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 that control neighbouring portions of that environment, with which it typically is strictly coordinated (Estrin *et al*., 2002).

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 a

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

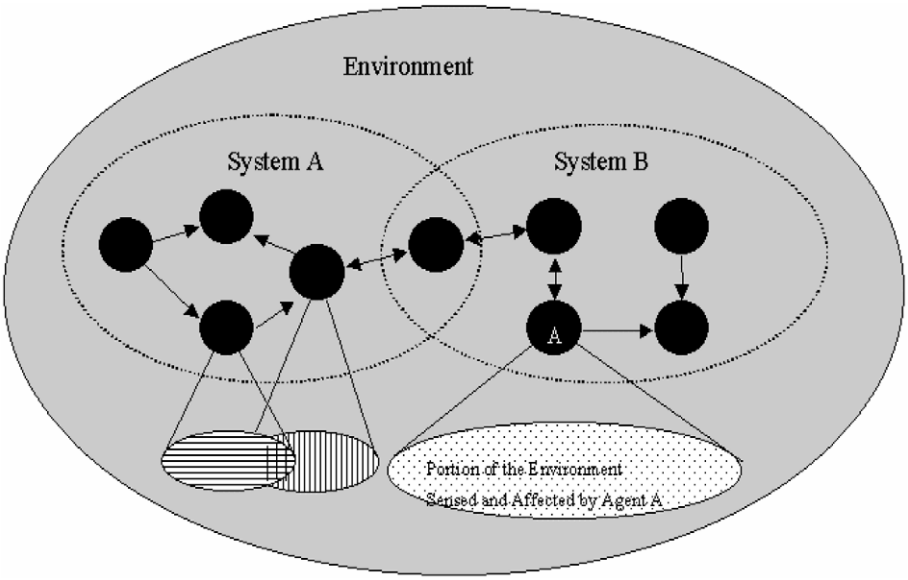


Figure 1 The scenario of modern software systems

limited range, a mobile computing component can directly interact only with a limited portion of the world at a given time (Picco *et al*., 2000; Broch *et al*., 1998).

Applications distributed on the Internet have to take into account the specific characteristics of the local administrative domain in which their components execute and have to interact, and components are usually allocated in Internet nodes so as to enforce locality in interactions as much as possible (Cabri *et al*., 2002; White, 1997).

2.5 A general model

In summary, software systems can be increasingly assimilated in 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 the components of other systems. Accordingly, some components may 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 of (or modelled 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 with each other, two components may interact indirectly with each other via the environment.

The scenario of Figure 1 is very different from that of component-based and object-based programming, and matches the prevailing model of agent-based computing (Jennings, 2001). Agents are considered as situated software entities (i.e. they live in an environment) that execute autonomously (i.e. have local control of their actions) and that 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 scientists working on agent-based computing still focus mostly on the ''artificial intelligence'' aspects of the discipline, without noticing (or simply deliberately ignoring) the fact that agents and agent-based computing have the potential to be a general model for today's computing systems, and that different research communities are facing similar problems and are starting to adopt similar modelling techniques. Similarly, outside the artificial intelligence community, computer scientists and software engineers fail to recognise that their current systems can be assimilated to agent-based systems and modelled as agent-based systems.

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

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

3 Changing our attitudes

Traditionally, software systems are modelled 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 analysing 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 a 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. Such a scenario may require untraditional models and methodology.

3.1 How will computer science change?

Modelling and handling systems with a very large number of components can be feasible if such components are not autonomous, i.e. they 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 so as to describe precisely the system's behaviour in terms of the behaviour of its components. In addition, as several software systems are distributed and subject to decentralised control, or possibly embedded in some unreachable environment (Estrin *et al*., 2002), tracking components and controlling their behaviour is simply impossible. Such systems can only be described and modelled 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 behaviour of a system will emerge not only as the sum of the behaviour of its components, but also as a result of their mutual interactions. One may argue that since interactions tend to be confined locally (Section 2.4), it should be quite easy to control their effect. Unfortunately, the effect of local interactions can propagate globally and be very difficult to predict (Prigogine & Steingers, 1991; Mamei *et al*., 2003). Propagation of local effects over a long distance is a hallmark of phase transitions in physical systems (Binney *et al*., 1992), to which many analogs are being discovered in distributed computational systems (Vattay, 2003). Even if one knows the initial status of the system very accurately, nonlinearities can amplify small deviations to become arbitrarily great, making prediction over a long time horizon impossible.

As an additional problem, we must consider that 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, changing the structure of the interaction graph. Thus, it is difficult to predict and control precisely not only the global dynamic behaviour of the system, but also how such behaviour can be influenced by such openness. In fact, it has been shown that the effects of interactions of autonomous active components strongly depend on the structure of the interaction graph (Watts, 1999), so that the dynamic behaviour of a system can change completely with only slight changes in the structure of the interaction graph.

Situatedness causes similar problems. In fact, modern thermodynamics and social science tell us that environmental forces can produce very strange and large-scale dynamic behaviours on situated physical, biological, and social systems (Prigogine & Steingers, 1991). We can expect the same large-scale effects to emerge on situated software systems, because of the dynamics of the environment, as a preliminary set of experiments suggest (Mamei *et al*., 2003).

The above problems will force computer scientists to change their attitudes dramatically in the modelling of complex software systems. Traditional formalisms and methods, aimed at proving the properties of software via logical tools, can deal effectively only with very small systems (or with

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

small portions of large systems). The dream of dealing with fully formalisable software systems has already started to vanish with the advent of concurrent and interactive systems (Wegner, 1997), and it will become even more impractical in the next few years. In fact, as soon as a system becomes large, open, and made up of autonomous and situated components, its behaviour can become so complex so as to be irreducible (Wolfram, 2002). The only way to understand the behaviour of such a system is to execute the system and observe it.

The next challenge is to find alternative models or, more radically, to adopt a brand-new scientific background for the study of software systems, enabling us to study, predict, and control the properties of a system and its dynamic behaviour (or at least some relevant properties and some specific kinds of observable behaviour) despite the inability to control and predict the micro-level behaviour of its individual components.

3.1.1 Signals

Some signals of this trend can already be found in different areas of the research community.

Recent study and monitoring activities on the Web (Crovella & Bestavros, 1997; Albert *et al.*, 1999) and on P2P networks (Ripeani *et al.*, 2002) have made it clear that unpredictable and large-scale behaviours are already here, requiring new models and tools for their description. For instance, it has been shown that traditional Web caching policies are no longer effective when peculiar dynamic Web-access patterns emerge (Crovella & Bestavros, 1997), and that traditional reliability models fall short due to the specific emergent characteristics of Web and P2P networks (Albert *et al.*, 2000; Ripeani *et al.*, 2002).

Some approaches to model and describe software systems in terms of thermodynamic systems have already been proposed (Parunak & Bruekner, 2001; Parunak *et al.*, 2002). The ideas behind such research are twofold: to provide synthetic indicators capable of measuring how closely the system is approaching the desired behaviour, and to provide tools to measure the influence of 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 system properties dynamically requires the introduction of synthetic indicators (Corradi *et al.*, 1999). Similarly, it has been recognised that modelling large and dynamic (overlay) networks can only be faced by introducing macro properties rather than by direct representation of the network (Albert *et al.*, 1999; Ripeani *et al.*, 2002).

One area that clearly shows such a trend is artificial intelligence. The claim that ''rational'' intelligence can emerge from a complex machine capable of manipulating facts and logic theories has always been strongly debated since the origins of computer science, and several alternative ways to model intelligence and to build intelligent systems in terms of large interactive systems have been proposed (e.g., neural networks and evolutionary algorithms). Nevertheless, all these proposals have only led to special-purpose applications, and never entered the mainstream of computer science and artificial intelligence. Now, the abstractions promoted by agent-based computing and multi-agent systems (matching the characteristics of today's software systems, Section 2.5) have promoted a shift to the concept of ''intentional'' intelligence, i.e. the capability of a component or of a system to behave autonomously and to interact so as to achieve a given, general-purpose, goal. In this context, organisation theory (Zambonelli *et al.*, 2003) and social science (Moses & Tenneholtz, 1995) are starting to influence research, as it is recognised that the behaviour of a large-scale software system can be assimilated more appropriately to a human organisation aimed at reaching a global organisational goal, or to a society in which the overall global behaviour derives from the self-interested intentional behaviour of its individual members, rather 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 the biological sciences (Huberman & Hogg, 1993; Gustavsson & Fredriecksson, 2001; Nagpal, 2002).

Given the above signals, we expect theories and models from complex dynamical systems and modern thermodynamics, as well as from biology, social science, and organisational science, to

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

become more and more the *sine-qua-non* cultural background of the computer scientist (other than, as of today, of researchers in the area of multi-agent systems).

3.2 How will software engineering change?

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

With regard to design, the lack of micro-level control in a software system makes it impossible to obtain a well-defined behaviour of a multi-component system ''by design'', i.e. in mechanical and deterministic terms. The next challenge for the effective construction of large software systems is to build them so as to guarantee that the system as a whole will behave as desired (or reasonably close to an ideal desired behaviour) despite the lack of exact knowledge about its micro-behaviour. For instance, by adopting a ''physical'' attitude towards software design, a possible approach could be to build a system that, despite uncertainty on the initial conditions, is within a basin of attraction leading to a stable attractor. 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, which is robust enough to direct its global activities towards the achievement of the required goal, or to approach the goal reasonably closely. The design must also explicitly take into account the fact that a system will be immersed in an open and dynamic environment, making it useless to design it in isolation. By promoting the environment and its dynamics to a primary design abstraction, the design may either assume a defensive approach (treating them as sources of uncertainty that can somehow be damaging to the global behaviour of a system and that the system should be prepared to face) or an offensive approach (considering openness and environmental dynamics as additional design dimensions to be exploited with the possibility of improving the behaviour of the system (Paranak *et al.*, 2002)).

Testing, traditionally, amounts to analysing system state transitions to see if they correspond to the designed ones or if, instead, some of the components exhibit bad state transitions (i.e. bugs). While it is already well known that such work is very hard for large (even non-concurrent) systems, it may become impossible when autonomy and environmental dynamics produce a practically uncountable number of states and non-deterministic state transitions. Thus, a large software system will no longer be tested with the goal of finding errors, but rather with regard to its ability to behave as needed as a whole, independently of the exact behaviour of its components and of their initial conditions (Huhns, 2001). Moreover, a software system that is likely to be immersed 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 greater its ability to 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).

Maintaining software will change too. First of all, because of autonomy and openness, every software system can be considered to be, to some extent, unstable and in continuous need of maintenance due to changed conditions. From a more traditional perspective on maintenance, when a large software system no longer behaves as needed (for instance, when the external conditions require some change in the behaviour of a software system), update will no longer imply stopping the system, rebuilding it, and retesting it. Instead, it will imply intervening on the system from the outside, by adding new components with different attitudes and by removing some of its existing components so as to change, as needed, the overall behaviour of the system. In this case, the openness and situatedness of the system and the autonomy of its component 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.

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

3.2.1 Signals

Again, it is possible to identify a few exemplary projects that are already adopting, to different extents, such a novel software engineering perspective.

In the area of distributed operating system 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, despite the dynamics of the environment (Cybenko, 1989). The design of these policies is, in several cases, inspired by physical phenomenon, such as diffusion. This perspective has proven useful to re-establish global equilibrium in dynamic situations (Corradi *et al*., 1999), despite the fact that such equilibrium will never be perfect but always locally perturbed.

Systems of ant colonies designed in a bottom-up way are able to solve very complex problems, which are hard to solve otherwise, via very simple autonomous components interacting in a dynamic environment (Parunak, 1997). There, the idea is to mimic in software the behaviour of insect colonies living in a dynamic world and able to solve, as a group, problems that have an interesting computational counterpart (i.e. sorting and routing). In that case, the environmental dynamics plays a primary role in the design and in the emergence of specific useful behaviours. In fact, in such systems, a fundamental phase of design and testing relates to the understanding and verification of how the environmental activities impact on the overall behaviour of the insect colony.

Inspiration from social phenomena such as epidemics has been useful in understanding distributed information in dynamic networks of components, such as mobile *ad hoc* networks (Picco *et al*., 2003), despite the inability to control the information paths and the structure of a network exactly. There, the dynamics of the network is both a source of uncertainty and a useful property to guarantee that a message can be propagated to the whole network in a reasonable time. P2P information dissemination systems exploit in a similar way the dynamic properties of a spontaneously generated community network (Ripeani *et al*., 2002), which can survive dynamics and changes in users' interests and in network structure without any sort of centralised management and without any user-directed maintenance. The related phenomenon of gossip, through which information can reach any person in a social network with dramatic speed, has recently inspired routing algorithms in different areas (i.e. wide-area event notification and routing in sensor networks (Braginsky & Estrin, 2002)).

The impossibility of detecting the structure of a network and of its components, because of both the dimension and the intrinsic dynamics of the network, is challenging the whole concepts of information routing and information retrieval. In different areas (e.g., pervasive and mobile computing (Adjie-Winoto *et al*., 1999), P2P Internet computing (Rowstron & Druschel, 2001; Ratsanamy *et al*., 2001; Stoica, 2001), middleware (Carzaniga *et al*., 2001), sensor networks (Intanagonwiway *et al*., 2000; Estrin *et al*., 2002)), there is a general understanding that the dynamics of networks require novel (content-oriented rather than path-oriented) approaches to information retrieval and routing. In other words, the basic idea is that paths to information sources/destinations should be found dynamically by relying on abstract overlay networks that continuously and automatically reshape themselves in response to system dynamics, and where data (or queries) can follow the shape of the overlay network just as a ball rolls down a sloped surface. The final destination is less important than the necessity that the paths followed by data and queries always proceed in a ''good'' direction and eventually reach a point where nodes interested in the routed message (or where interesting information) can be found. It is also worth noting that such systems are typically tested and evaluated in terms of how much the system can tolerate network dynamics by still exhibiting reasonably good behaviours, how fast the overlay network can re-organise in response to dynamic changes, and how the addition or removal of components impact on the behaviour of the network (Albert *et al*., 2000; Rowstron & Druschel, 2001).

A biological phenomenon such as evolution, despite its intrinsic uncertainty, is also likely to become a useful tool for software engineers. For instance, although cellular automata have the potential to perform complex computations as a result of their dynamic evolution, it turns out to

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

be almost impossible to understand which rules must be imposed on cells and on their interaction so as to produce a system with certain required properties (Wolfram, 2002). 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 behaviour of the automata, without anyone having directly designed such behaviours (Sipper, 1999).This approach has also been successful with computational systems more complex than cellular automata (Sauter *et al*., 2002).

4 Discussion and 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 on the very way software systems will be modelled and engineered, leading to a paradigm shift in computer science and software engineering (Kuhn, 1996). 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, and consequently a brand new set of conceptual tools and methodologies.

One possible criticism of this approach is that software systems, on which humans are becoming more and more dependent for their everyday activities and for the control of safe critical situations, cannot be engineered in the way we have traditionally envisioned. Many claim that a software system must exhibit predictable and fully controllable behaviour in each of its parts, and that the duty of a software engineer is to produce such reliable systems. For instance, pessimists foresee the sudden death of peer-to-peer software systems, despite their current success, because of their unreliability and the impossibility of properly modelling them. In our opinion, this approach is not correct. Constraining the behaviour of a highly interactive system may sacrifice most of its computational power and may require a much greater 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 make a useful global behaviour, still reliable and stable, emerge from it. Finally, constraining a software system ''by design'' may simply make it lose the properties needed to support openness and to tolerate environmental dynamics.

Despite the opposing forces and the difficulties inherent in any revolution, including the need to restructure our cultural background, the envisioned revolution in computer science and software engineering will definitely open the door to new interesting research and engineering challenges, and will be likely to enable new powerful applications of ICT technologies.

References

Abelson, H. *et al*. 2000 Amorphous computing. *Communications of the ACM* **43 (5), 43–50.**

Abowd, G. D. & Mynatt, E. D. 2000 Charting past, present and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction* **7**(1), 29–58.

Adjie-Winoto, W., Schwartz, E., Balakrishna, H. & Lilley, J. 1999 The design and implementation of an intentional naming system. *ACM Operating Systems Review* **34**(5), 186–201.

Albert, R., Jeong, H. & Barabasi, A. 1999 Diameter of the World Wide Web. *Nature* **401**, 130–131.

Albert, R., Jeong, H. & Barabasi, A. 2000 Error and attack tolerance of complex networks. *Nature* **406**, 378–382.

Binney, J. J., Dowrick, N. J., Fisher, A. J. & Newman, M. E. J. 1992 *The Theory of Critical Phenomena – An Introduction to the Renormalization Group*. Oxford: Clarendon Press.

Braginsky, D. & Estrin, D. 2002 Rumor routing algorithm for sensor networks. In *Proceedings of the 1st International Workshop on Sensor Networks and Applications, Atlanta, GE*. ACM Press, pp. 22–31.

Broch, J., Maltz, D. A., Johnson, D. B., Hu, Y. C. & Jetcheva, J. 1998 A performance comparison of multi-hop wireless ad-hoc network routing protocols. In *Proceedings of the 4th ACM Conference on Mobile Computing and Networking, Dallas, TX*. ACM Press, pp. 85–97.

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

Bussmann, S. 2000 Self-organizing manufacturing control: an industrial application of agent-technology. In *Proceedings of the 4th IEEE International Conference on Multiagent Systems, Boston, MA*. IEEE CS Press, pp. 87–94.

Cabri, G., Leonardi, L. & Zambonelli, F. 2002 Engineering mobile agent applications via context-dependent coordination. *IEEE Transactions on Software Engineering* **28**(9), 1041–1057.

Capra, F. 1997 *The Web of Life: The New Understanding of Living Systems*. New York: Doubleday.

Carzaniga, A., Rosenblum, D. S. & Wolf, A. L. 2001 Design and evaluation of a wide-area event-notification service. *ACM Transactions on Computer Systems* **19**(3), 332–383.

Corradi, A., Leonardi, L. & Zambonelli, F. 1999 Diffusive load balancing policies for dynamic applications. *IEEE Concurrency* **7**(1), 22–31.

Crovella, M. & Bestavros, A. 1997 Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking* **5**(6), 835–846.

Cybenko, G. 1989 Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing* **7**(2), 279–301.

Estrin, D., Culler, D., Pister, K. & Sukjatme, G. 2002 Connecting the physical world with pervasive networks. *IEEE Pervasive Computing* **1**(1), 59–69.

Gellersen, H. W., Schmidt, A. & Beigl, M. 2002 Multi-sensor context-awareness in mobile devices and smart artefacts. *ACM Journal on Mobile Networks and Applications* **7**(5), 341–351.

Gustavsson, R. & Fredriksson, M. 2001 Coordination and control in computational ecosystems: a vision of the future. In Omicini, A. *et al*. (eds.), *Coordination of Internet Agents*. Berlin: Springer, pp. 443–469.

Howard, A., Mataric, M. J. & Sukhatme, G. S. 2002 An incremental self-deployment algorithm for mobile sensor networks. *Autonomous Robots* **13**(2), 113–126.

Huberman, B. A. & Hogg, T. 1993 The emergence of computational ecologies. In *Lectures in Complex Systems*. Reading, MA: Addison-Wesley.

Huhns, M. 2001 Interaction-oriented programming. In *Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering* (*Lecture Notes in Computer Science, Vol. 1957*). Berlin: Springer-Verlag.

Intanagonwiway, C., Govindam, R. & Estrin, D. 2000 Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th ACM/IEEE Conference on Mobile Computing and Networking, Boston, MA*. ACM Press, pp. 56–67.

Jennings, N. R. 2001 An agent-based approach for building complex software system. *Communications of the ACM* **44**(4), 35–41.

Kuhn, T. 1996 *The Structure of Scientific Revolutions*, 3rd edn. Chicago, IL: University of Chicago Press.

Mamei, M., Roli, A. & Zambonelli, F. 2003 Dissipative cellular automata as minimalist distributed systems: a study on emergent behaviours. In *Proceedings of the 11th EUROMICRO Conference on Parallel, Distributed, and Network Processing, Genova*. IEEE CS Press.

Moses, Y. & Tenneholtz, M. 1995 Artificial social systems. *Computers and Artificial Intelligence* **14**(3), 533–562.

Nagpal, R. 2002 Programmable self-assembly using biologically inspired multiagent control. In *Proceedings of the 1st International Conference on Autonomous Agents and Multiagent Systems, Bologna*. ACM Press, pp. 418–425.

Nagpal, R., Shrobe, H. & Bachrach, J. 2003 Organizing a global coordinate system from local information on an ad hoc sensor network. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks, Palo Alto, CA*. ACM Press.

Odell, J., Parunak, H. V. D., Fleischer, M. & Brueckner, S. 2003 Modeling agents and their environment. In *Proceedings of the 3rd International Workshop on Agent-Oriented Software Engineering* (*Lecture Notes in Computer Science, Vol. 2585*). Berlin: Springer-Verlag, pp. 16–31.

Parunak, H. V. D. 1997 Go to the ant: engineering principles from natural agent systems. *Annals of Operations Research* **75**, 69–101.

Parunak, H. V. D. & Bruekner, S. 2001 Entropy and self-organization in agent systems. In *5th International Conference on Autonomous Agents, Montreal, CA*. ACM Press, pp. 124–130.

Parunak, H. V. D., Bruekner, S. & Sauter, J. 2002 ERIM's approach to fine-grained agents. In *NASA/JPL Workshop on Radical Agent Concepts, Greenbelt, MD*. January 2002.

Picco, G. P., Cugola, G. & Murphy, A. L. 2003 Efficient content-based event dispatching in presence of topological reconfiguration. In *Proceedings of the 23rd International Conference on Distributed Computing Systems, Providence, RI*. IEEE CS Press.

Picco, G. P., Murphy, A. M. & Roman, G.-C. 2000 Software engineering for mobility: a roadmap. In Finkelstein, A. (ed.), *The Future of Software Engineering*. ACM Press, pp. 241–258.

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

Priyantha, N. B., Miu, A. K. L., Balakrishnan, H. & Teller, S. 2001 The cricket compass for context-aware mobile applications. In *Proceedings of the 7th ACM/IEEE Conference on Mobile Computing and Networking, Rome*. ACM Press, pp. 1–14.

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press

Ratsanamy, S., Francis, P., Handley, M. & Karp, R. 2001 A scalable content-addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference, San Diego, CA*. ACM Press.

Ricci, A., Omicini, A. & Denti, E. 2003 Activity theory as a framework for MAS coordination. In *Proceedings of the 3rd International Workshop on Engineering Societies in the Agents World* (*Lecture Notes in Artificial Intelligence, Vol. 2577*). Berlin: Springer.

Ripeani, M., Iamnitchi, A. & Foster, I. 2002 Mapping the Gnutella network. *IEEE Internet Computing* **6**(1), 50–57.

Rodriguez-Aguilar, J. A. & Sierra, C. 2002 Enabling open agent institutions. In K. Dautenhahn *et al.* (eds.), *Socially Intelligent Agents: Creating Relationships with Computers and Robots*. New York: Kluwer Academic Publishers, pp. 259–266.

Rowstron, A. & Druschel, p. 2001 Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM Conference on Distributed Systems Platforms, Heidelberg*, November 2001.

Sauter, J. A., Matthews, R., Parunak, H. V. D. & Brueckner, S. 2002 Evolving adaptive pheromone path planning mechanisms. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna*. ACM Press, pp. 434–440.

Sipper, M. 1999 The emergence of cellular computing. *IEEE Computer* **37**(7), 18–26.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. & Balakrishnan, H. 2001 Chord: a scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference, San Diego, CA*. ACM Press.

Tolksdorf, R. 2000 Coordinating work on the web with workspaces. In *Proceedings of the 9th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, Gaithersburg*. IEEE CS Press.

Tennenhouse, D. 2000 Proactive computing. *Communications of the ACM* **43**(5), 43–50.

Vattay, G. 2003 *The Internet Physics Web Page*, `http://galahad.elte.hu/Nvattay/Internetphysics.htm`

Watts, D. 1999 *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton, NJ: Princeton University Press.

Wegner, P. 1997 Why interaction is more powerful than algorithms. *Communications of the ACM* **40**(5), 80–91.

Weiser, M. 1993 Hot topics: ubiquitous computing. *IEEE Computer* **26**(10), 71–72.

White, J. 1997 Mobile agents. In J. Bradshaw (ed.), *Software Agents*. Menlo Park, CA: AAAI Press, pp. 437–472.

Wolfram, S. 2002 *A New Kind of Science*. New York: Wolfram Inc.

Wooldridge M. J. & Jennings, N. R. 1998 Pitfalls of agent-oriented development. In *Proceedings of the 2nd International Conference on Autonomous Agents, Minneapolis, MN*. ACM Press, pp. 385–391.

Zambonelli, F., Jennings, N. R. & Wooldridge, M. J. 2003 Developing Multiagent Systems: the Gaia Methodology. *ACM Transactions on Software Engineering and Methodology* **12**(3).

https://doi.org/10.1017/S0269888904000104 Published online by Cambridge University Press