Volume 2, Issue 7, July 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Computer Science and Software Engineering: A Change in Archetype

K. Kalyani
Asst.Professor
Dept. Of Computer Science
Matrusri Inst. of PG Studies
Hyderabad, A.P., India

***Abstract*— A set of characteristics that increasingly distinguish today's complex software systems from "traditional" ones. We identify and analyze these by several examples in different areas show that these characteristics are not limited to a few application domains but are widespread. These characteristics are likely to impact dramatically the way software systems are modeled and engineered. In particular, we appear to be on the edge of a radical shift of paradigm, about to change the attitudes in software systems modeling and engineering.**

***Keywords*—Distributed Systems, Design Paradigms, Multiagent Systems, Signals, Future.**

I. INTRODUCTION

Computer science and software engineering are going to change dramatically. Computer Science is the study of computer systems including algorithmic processes and the principles involved in the design of hardware and software where as Software Engineering is the practice of designing and implementing large, reliable, efficient and economical software by applying the principles and practices of engineering.

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 situation initiating the next software crisis is rapidly emerging [1]. computing systems will be everywhere, always connected, and always active Computer systems will be embedded in every object[5], like in our physical environments, our clothes and furniture, and even our bodies. Wireless technologies will make network connectivity persistent , 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 for example to improve comfort at home or to control and automate manufacturing processes.[7].This situation 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 addition to the quantitative increase in interconnected computer systems four main characteristics distinguish future software systems from the traditional ones.

***Situations*:** 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*:** Inspite of living in a fully connected world, software components interact with each other accordingly to local (geographical or logical) patterns.

The above four characteristics are the ones that are typically used to characterize multi-agent systems in the research community of distributed artificial intelligence [6]. . we can see that several research communities focused on different topics with the above characteristics such as manufacturing and environmental control systems mobile and pervasive computing Internet andP2P computing are already recognizing their importance and are already adapting their models and technologies to take them into account. Thus, our first contribution is to attempt at synthesizing in a single conceptual framework several novel concepts and abstractions that are emerging in different areas without recognition of the basic commonalties, as there is a lack of interaction and common terminology.

The integration of these concepts and abstractions in software modeling and design does not simply represent a proper evolution of current models and methodologies.

Instead, we look forward a real revolution or a scientific change of prototype, likely to impact most research communities horizontally and to change the way we consider, model, and build, "software components" and "software systems". The signals in the next generation that testify software systems will no longer be modeled and designed in terms of "mechanical" or "architectural" systems, but rather in terms of "physical" or "intentional" systems.

II The Change in Computer Science and Software Engineering

Computer science is becoming a rapidly growing discipline as the technological age advances. Computer scientists believe that computers are a fundamental part of the world and that an age will come when everybody has several computers. It is a more complex field than simply building computers or writing programs. Computer scientists study problems to determine if they can be computed, compare algorithms to decide on the best solution, create programming languages to express these algorithms, design and build computer systems to execute specifications from research, and apply algorithms to application domains, or sets of software systems that share design features where as Software engineering is the computer science discipline concerned with developing large applications. Software engineering covers not only the technical aspects of building software systems, but also management issues, such as directing programming teams, scheduling, and budgeting.

III. THE CHARACTERISTICS

The four characteristics identified in the introduction are situations, openness, local control, local interactions affect, with different flavors and to different extents, to most of today's software systems.

Situations

The computing systems are situated in an explicit notion of the environment
in which components are allocated and execute and are affected in their execution by
environmental characteristics, and their components often explicitly interact with that
environment.

We accentuate that software systems have never worked in isolation but immersed in some sort of environment. For example, the execution of a running process in a multi-threaded machine is basically affected by the dynamics of the multiprocessing system. However, traditional modeling and engineering approaches have always tried to cover the presence of the environment. In most cases, specific objects and components bind 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 indeed exists and may impact on application execution and on modelling the application as:

- The software components possibly interact are too complex in their structure and behavior to enable a trivial wrapping.
- The goal of a software system is to monitor and control a physical or logical environment that is not a natural choice but providing alternatively explicit perception becoming a primary application goal.
- Casing the environment in an application entity will begin forms of unpredictable non-determinism inside applications as the software systems are independent of its own dynamics.

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[3].This approach both avoids odd covering 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 and also the explicit modeling of the environment and of its activities makes it possible to recognize and detain clearly the sources of dynamics.

Examples.

Control systems for physical domains like manufacturing, traffic control, home care tend to be built by explicitly managing environmental data, and by explicitly taking into account the unpredictable dynamics of the environment by means of specific event-handling policies[3]. Mobile and pervasive computing applications recognize the primary importance of context awareness, as the need for applications to maintain explicit models of environmental characteristics like characteristics related to the sensed physical environment 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 that include the new application components to be developed and "middleware" [4] in which components will be Embedded.

It is worth noting that several proposals in the area of distributed problem solving and optimization are based on a model centered around a dynamic virtual environment influencing the activities of distributed problem solver processes.

Openness

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.

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. These are independently designed and modelled in the same environment to interact with each other. These are open interactions for common communication protocols and suitable broker infrastructure to enable interoperability. Yet, 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.

These characteristics introduce other problems:

- When a component interacts with other components in other software systems, and when components move across different environment, it becomes

difficult to understand the boundaries of software systems clearly.

- When a component comes to life in an environment must be made aware of its environmental context, what other components for interaction, and how it can interact in the newly entered system
- 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.

Due to the above problems, software development may require facing the problem of not only modeling the environment in which systems execute, but also of understanding and modeling dynamic changes in the system structure.

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. Hence these systems need to be updated continuously with the environment[1]. For all these systems,

managing openness and the capability of a system to re-organize itself automatically

Mobility, whether of users, software, or devices, moves the concept of openness to the

extreme, by making components actually move from one context [4]to another during their

execution, thus changing the structure of the software system executing in that context

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 like admin domain, Internet applications etc.

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. Yet, 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 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. Most of the

software systems are active entities like active objects with internal threads of control,

processes, daemons rather than passive ones like "normal" objects, functions, etc

We are known that concurrent and parallel programming are well-established paradigms, and applications with multiple threads and processes have been around for a long time. But traditional concurrent and parallel programming need to improve efficiency and performance, develop multiple flows of execution by avoiding making them multiple threads of control. But most traditional approaches limit as much as possible the autonomy of these multiple flows of execution, through synchronization and coordination mechanisms in execution the application. This tendency can be seen even in some approaches to multiagent systems engineering [4].The application components has 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 acknowledgement to wait for its application.
- When components and systems are wrapped in a highly dynamic environment whose evolution must be monitored and controlled, an autonomous component can be effectively delegated to take care the environment independently for the flow of control.
- Several software systems are not only made up of software components, but also integrate computer-based system that can be modeled accordingly.
- he need of delegating control to components is no longer simply a matter of performance As the size of a software system increases, but becomes a matter of conceptual simplicity. Infact, coordinating a global flow of control among a large number of components may become unfeasible, and an additional dimension of modularity[9]

Examples.

In 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 of graphical interfaces or simple embedded sensors. Autonomy implies
that a component integrates thread of execution that can execute in a practical way.
This is the case in most modern control systems for physical domains, which control is not simply reactive but practical, realized through a set of cooperative autonomous processes or through distributed sensor networks [7]. .
The integration in complex distributed applications and software running on
mobile devices can be tackled only by modeling them in terms of autonomous software components[4]. .
Internet based distributed applications are typically made up of autonomous processes,
possibly executing on different nodes by decentralized management rather than by the actual need of autonomous concurrent activities.

Local Interactions

The concept of "local interactions in a global world" is more and more pervading software systems. We have delineated a situation that software system components are wrapped in a specific environment, execute in the context of a specific system to
perform some task autonomously. By taking all these aspects lead to a strict
enforcement of locality in interactions. In fact:

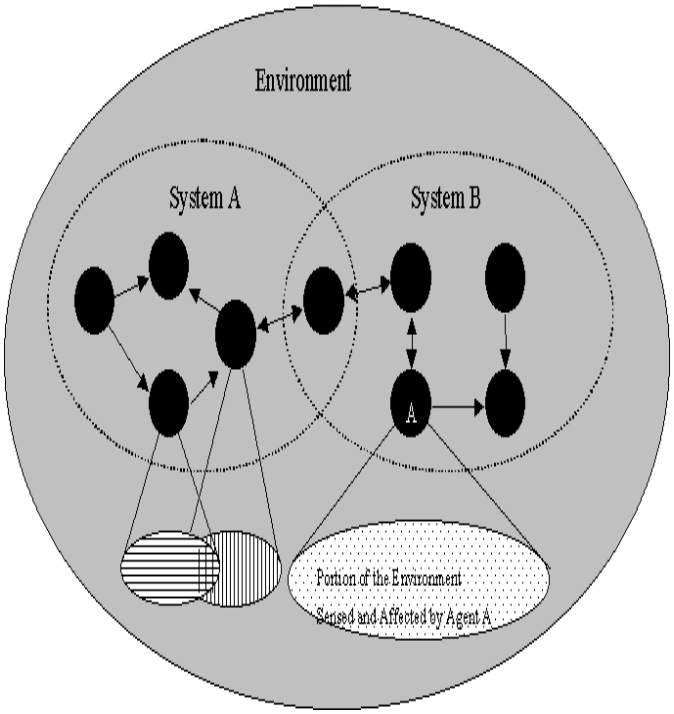
- Autonomous components can interact with the environment in which they are wrapped by sensing and affecting it.
  If the environment is physical, a single component can sense and affect can be locally bounded by physical laws. If the environment is logical, minimization of conceptual and management complexity still favors modeling in limiting the effect of a single component on the environment.
- The conceptual complexity suggests modeling the component temporarily moved to another context to interact locally[4]. if components can normally interact with each other in the context of the software system locally to their system.

- 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. In an open world, components need some form of context-awareness to execute and interact

Examples.

Control systems for physical domains tend to implement 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 limited [5].to those that control neighbouring portions of that environment.

In mobile computing, the 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 only with a limited portion of the world at a given time[5].
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 [4].



The Modern software systems Scenario

A General Model

In some software systems can be increasingly assimilated in the scheme of Figure 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. Some components may be part of several software systems at different times,

depending on their current interaction activities. Systems and components are wrapped in an environment, composed of 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 through the environment.

The picture of Figure is very different from that of component-based and object-based programming, and matches the existing model of agent-based computing[6].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. Even thought, most scientists working on agent-based computing still focus mostly on the .artificial intelligence. Aspects of the discipline, without noticing that agents and agent-based computing have the possible to be a general model for today's computing systems and that different research communities are facing similar problems and are starting adopting similar modeling techniques. Similarly, outside the artificial intelligence community, computer scientists and software engineers fail to recognize that their current systems can be assimilated to agent-based systems and modeled as agent-based systems.

The issues discussed in this can clarify these strict relationships and the need for more interaction between different research communities, due to the strong similarities between the problems they address.

IV. CHANGING OUR ATTITUDES

Traditionally, software systems are modeled from a mechanical posture, and engineered from a design posture. Computer scientists are both hampered and enthralled 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 display in a top-down way, and to designing software architectures as trustworthy multi-component machines, capable of providing the required functionality competently and Inevitably.

Scientists and engineers will have to design software system to execute in a world where a multitude of autonomous, embedded, and mobile software components that are already executing and interacting with each other and with the environment on the basis of local interaction patterns in the future. Such a scenario may require untraditional models and methodology.

Changes in Computer Science

Modeling and handling systems with a very large number of components can be feasible if such components are not autonomous, that is 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 accurately the system's behavior in terms of the behavior of its components. In addition, as several software systems are distributed and subject to decentralized control, or embedded in some unreachable environment 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 visible 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 some of the behavior of its components, but also as a result of their mutual interactions. One may argue that since interactions tend to be confined locally (Subsection 2.4) it should be quite easy to control their effect. Regrettably, the effect of local interactions can spreade globally and be very difficult to predict[8]. Propagation of local effects over a long distance is a brand of phase transitions in physical systems [2]to which many analogs are being discovered in distributed computational systems[3]. Yet if one knows the initial status of the system very perfectly, 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 envisage and control accurately 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 depend on the structure of the interaction graph so that the dynamic behavior of a system can change completely with only slight changes in the structure of the interaction graph.

Situations causes similar problems. In fact, modern thermodynamics and social science tellus that environmental forces can produce very strange and large scale dynamic behaviors on situated physical, biological, and social systems [4]. We can expect the same large-scale effects to appear on situated software systems, because of the dynamics of the environment, as a preliminary set of experiments suggest[6].

The above problems will force computer scientists to change their attitudes dramatically in the modeling of complex software systems. Traditional formalisms and methods, aimed at proving the properties of software through logical

tools, can deal effectively only with very small systems. The dream of dealing with fully formalizable
software systems[1] has already started to disappear with the advent of concurrent and interactive systems 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 behavior can become so complex to be irreducible[4] .The only way to understand the behavior of such a system is to execute the system and observe it.
The next challenge is to find alternative models or, more thoroughly, to adopt a brand-new scientific background for the study of software systems, enabling us to study, envisage, and organize the properties of a system and its dynamic behavior or at least some specific kinds of observable behavior even though the inability to control and envisage the micro-level behavior of its individual components.

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[5] and on P2P networks [3] have made it clear that unpredictable and large-scale behaviors are already here, enquiring 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 come out and that traditional reliability models fall short due to the specific emergent characteristics of Web and P2P networks [6].
Some approaches to model and describe software systems in terms of thermodynamic
systems have already been proposed[3].. The ideas behind such research are
twofold: to provide artificial indicators capable of measuring how closely the system is
approaching the desired behavior, 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 especially parallel computing, where the need to measure specific global systems properties dynamically requires the introduction of artificial indicators. 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[3].
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 such as neural networks and evolutionary algorithms. Nevertheless, all these proposals led only to special-purpose applications, and never entered the mainstream of computer science and artificial intelligence. Now, the abstractions promoted by agent-based computing and multiagent systems 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, organization theory and social science 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 the biological sciences [7].
Given the above signals, we expect theories and models from complex dynamical systems and modern thermodynamics, as well as from biology, social science, and organizational science, to become more and more the sine-qua-non cultural background of the computer scientist .

Changes in Software Engineering

The change in the modeling 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 behavior of a multi-component system by design,. i.e., in mechanical and deterministic terms. The next challenge for the effectual construction of large software systems is to build them so as to guarantee that the system as a whole will behave as desired in spite of the lack of exact knowledge
about its micro-behavior. For example, 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 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 deliberate way, and strong enough to direct its global activities toward 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 separation. By promoting the environment and its dynamics to a
primary design abstraction, the design may either assume a defensive approach
or an offensive approach by considering openness and environmental dynamics as additional design dimensions to be exploited with the possibility of improving the behavior of the system[9]).

Testing, traditionally, amounts to analyzing 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 not 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 behavior of its components and of their initial conditions. Moreover, a software system that is likely to be wrapped up 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 target. 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 strong 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 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 behavior of a software system update will no longer involve 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 behavior of the system. In this case, the openness and situations 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 put up with and support dynamic changes in its structure.

Signals. Yet again, it is possible to identify a few excellent projects that are already adopting, to different extents, such a novel software engineering perspective.

In the area of distributed operating systems management, policies for the management of distributed resources are already being designed in terms of autonomous components able to guarantee the achievement of a global goal via local actions and local interactions, in spite of the dynamics of the environment[6] . The design of these policies is, in several cases, motivated by physical phenomenon, such as diffusion. This perspective has proven useful to re-establish global balance in dynamic situations[in spite of the fact that such balance will never be perfect but always locally troubled.

Systems of ant colonies designed in a bottom-up way are able to solve very complex
problems, which are hard to solve otherwise, through a very simple autonomous components interacting in a dynamic environment [4].There, the idea is to take off in software the behavior 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 design and in the emergence of specific useful behaviors. In fact, in such systems, a fundamental phase of design and testing relates to understand and verify howthe environmental activities impact on the overall behavior of the insect colony.

Inspiration from a social phenomenon like epidemics has been useful in understanding
distributed information in dynamic networks of components, such as mobile ad-hoc networks [6],in spite of 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 giving out systems develop in a similar way the dynamic properties of spontaneously generated community network which can survive dynamics and changes in users. interests and in network structure without any sort of centralized management and without any user-directed maintenance. The related phenomenon of rumor, through which information can reach any person in a social network with dramatic speed, has recently inspired routing algorithms in different areas like wide area and routing in sensor networks [7]).

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 such as pervasive and mobile computing [3],P2PInternet[9], computing ,middleware [1],
sensor networks[7]), there is a general understanding that the dynamics of networks require novel ie(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 cover network just as a ball rolls down a sloped surface. The final destination is less important than that 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 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 stand 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 or removal of components
impact the behavior of the network[6].

A biological phenomenon such as evolution, in spite of its intrinsic uncertainty, is likely to
become a useful tool too for software engineers. For instance, though cellular automata have the potential to perform complex computations as a result of their dynamic evolution, it turns out to 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[4]. An approach that has been successfully experienced is to make cellular automata rules evolve by using 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. This approach has also been successful with computational systems more complex than cellular automata.

V . 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 adapted. These characteristics are likely to impact dramatically the very way software systems will be modeled and engineered, leading to a pattern shift in computer science and software engineering[4]. 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 denigration of this approach is that software systems, on which humans are 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 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 make a useful global behaviour, still reliable and stable, emerge from it.

Finally, constraining .by design. a software system may simply make it lose the properties needed to support openness and to tolerate environmental dynamics.

In spite of the opposing forces and the difficulties inherent in any revolution, including the need of reformation of 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

1 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) pp 43-50, May 2000
2 J. J. Binney, N. J. Dowrick, A. J. Fisher, M. E. J. Newman. *The Theory of Critical*
Phenomena - An Introduction to the Renormalization Group. Clarendon Press
(Oxford, UK), 1992.
3 R. Albert, H. Jeong, A. Barabasi, .Diameter of the World Wide Web., *Nature*, 401
pp130-131, 9 Sept. 1999.
4 F. Capra, *The Web of Life: The New Understanding of Living Systems*, Doubleday
(New York, NY), Oct. 1997.
5 D. Estrin, D. Culler, K. Pister, G. Sukjatme, .Connecting the Physical World with
Pervasive Networks., *IEEE Pervasive Computing*, 1(1) pp 59-69, Jan. 2002.
6 R. Jennings, "An Agent-Based Approach for Building Complex Software System",
Communications of the ACM, 44(4) pp35:41, 2001.
7 R. Nagpal, .Programmable Self-Assembly Using Biologically Inspired Multiagent
Control., *Proceedings of the 1st International Conference on Autonomous Agents and*
Multiagent Systems, Bologna (I), ACM Press, pp 418-425, July 2002.
8 N.B. Priyantha, A.K.L. Miu, H. Balakrishnan, S. Teller, .The Cricket Compass for
Context-aware Mobile Applications., *Proceedings of the 7th ACM/IEEE Conference*
on Mobile Computing and Networking, Rome (I), ACM Press, pp 1-14, July 2001
9 S. Ratsanamy, P. Francis, M. Handley,R. Karp, .A Scalable Content-Addressable
Network. *Proceedings of the 2001 ACM SIGCOMM Conference*, San Diego (CA),
ACM Press, Aug. 2001.

AUTHOR'S PROFILE

Mrs. K. Kalyani

has completed her M.Tech in Computer Science &Engineering . She is having 9years of experience in teaching. She is working as an Assistant Professor with Matrusri Institute Of PG Studies, Saidabad, Hyderabad.

