AN AGENT-BASED APPROACH FOR BUILDING COMPLEX SOFTWARE SYSTEMS

WHY AGENT-ORIENTED APPROACHES ARE WELL SUITED FOR DEVELOPING COMPLEX, DISTRIBUTED SYSTEMS.

NICHOLAS R. JENNINGS

Building high-quality, industrial-strength software is difficult. Indeed, it has been argued that developing such software in domains like telecommunications, industrial control, and business process management represents one of the most complex construction tasks humans undertake. Against this background, a wide range of software engineering paradigms have been devised. Each successive development either claims to make the engineering process easier or promises to extend the complexity of applications that can feasibly be built. Although evidence is emerging to support these claims, researchers continue to strive for more effective techniques. To this end, this article will argue that analyzing, designing, and implementing complex software systems as a collection of interacting, autonomous agents (that is, as a multiagent system [4]) affords software engineers a number of significant advantages over contemporary methods. This is not to say that agent-oriented software engineering represents a silver bullet [2]—there is no evidence to suggest it will represent an order of magnitude improvement in productivity. However, the increasing number of deployed applications [4, 8] bears testament to the potential advantages that accrue from such an approach.

In seeking to demonstrate the efficacy of agent-oriented techniques, the most compelling argument would be to quantitatively show how their adoption improved the development process in a range of projects. However, such data is simply not available (as it is not for approaches like patterns, application frameworks, and componentware). Given this situation, the best that can be achieved is a qualitative justification of why agent-oriented approaches are well suited to engineering complex, distributed software systems.

Managing Complexity in Software Systems

Industrial-strength software is complex: it has a large number of parts that have many interactions [9]. Moreover this complexity is not accidental [2], it is an innate property of large systems. Given this situation, the role of software engineering is to provide structures and techniques that make it easier to handle complexity. Fortunately for designers, this complexity exhibits a number of important regularities [9]:

• Complexity frequently takes the form of a hierarchy. That is, a system composed of interrelated subsystems, each of which is in turn hierarchic in structure, until the lowest level of elementary subsystem is reached. The precise nature of these organizational relationships varies between subsystems, however, some generic forms (such as client/server, peer, team, and so forth) can be identified. These

relationships are not static: they often vary over time.
• The choice of which components in the system are primitive is relatively arbitrary and is defined by the observer's aims and objectives.
• Hierarchic systems evolve more quickly than nonhierarchic ones of comparable size (that is, complex systems will evolve from simple systems more rapidly if there are clearly identifiable *stable intermediate forms* than if there are not).
• It is possible to distinguish between the interactions among subsystems and those within subsystems. The latter are both more frequent (typically at least an order of magnitude more) and more predictable than the former. This gives rise to the view that complex systems are *nearly decomposable*: subsystems can be treated almost as if they are independent, but not quite since there are some interactions between them. Moreover, although many of these interactions can be predicted at design time, some cannot.

Drawing these insights together, it is possible to define a canonical view of a complex system (see Figure 1). The system's hierarchical nature is expressed through the "related to" links, components within a subsystem are connected through "frequent interaction" links, and interactions between components are expressed through "infrequent interaction" links.

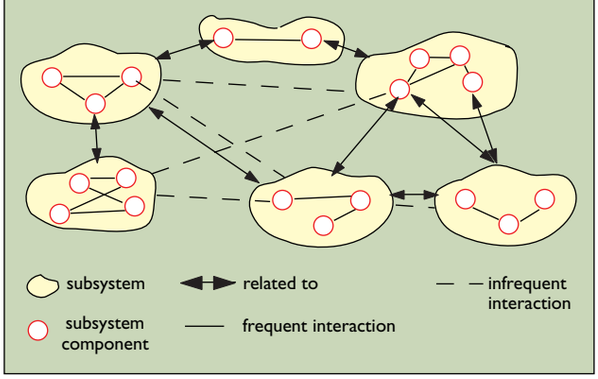
Given these observations, software engineers have devised a number of fundamental tools of the trade for helping to manage complexity [1]:

Decomposition: The most basic technique for tackling large problems is to divide them into smaller, more manageable chunks, each of which can then be dealt with in relative isolation (note the nearly decomposable subsystems in Figure 1). Decomposition helps tackle complexity because it limits the designer's scope.
Abstraction: The process of defining a simplified model of the system that emphasizes some of the details or properties, while suppressing others. Again, this works because it limits the designer's scope of interest at a given time.
Organization[1]**:** The process of defining and managing the interrelationships between the various problem-solving components (note the subsystem and interaction links of Figure 1). The ability to specify and enact organizational relationships helps designers tackle complexity by: enabling a number of basic components to be grouped together and treated as a higher-level unit of analysis, and providing a means of describing the high-level relationships between various units.

The Case for Agent-Oriented Software Engineering

The first step in arguing for an agent-oriented approach to software engineering involves identifying the key concepts of agent-based computing. The first such concept is that of an agent: an agent is an encapsulated computer system situated in some environment and capable of flexible, autonomous action in that environment in order to meet its design objectives [10].

Figure 1. View of a canonical complex system.



There are a number of points about this definition that require elaboration. Agents are: clearly identifiable problem-solving entities with well-defined boundaries and interfaces; situated (embedded) in a particular environment over which they have partial control and observability—they receive inputs related to the state of their environment through sensors and they act on the environment through effectors; designed to fulfill a specific role—they have particular objectives to achieve; autonomous—they have control both over their internal state and over their own behavior; capable of exhibiting flexible problem-solving behavior in pursuit of their design objectives—being both reactive (able to respond in a timely fashion to changes that occur in their environment) and proactive (able to opportunistically adopt goals and take the initiative) [11].

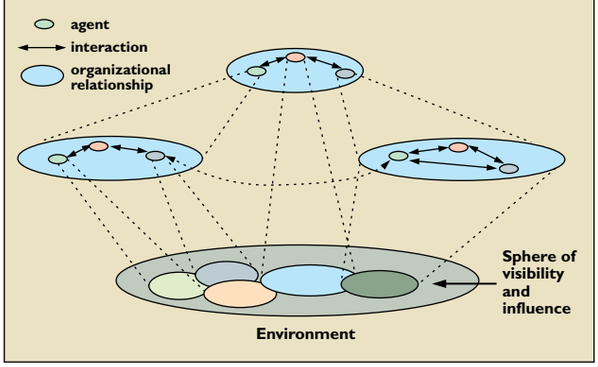
When adopting an agent-oriented view, it soon becomes apparent that most problems require or involve multiple agents: to represent the decentralized nature of the problem, the multiple loci of control, the multiple perspectives or the competing interests. Moreover, the agents will need to interact with one another: either to achieve their individual objectives

[1]Booch actually uses the term "hierarchy" [1]; however, this invariably gives the connotation of control. Hence the more neutral term "organization" is used here.

or to manage the dependencies that ensue from being situated in a common environment. These interactions can vary from simple semantic interoperation, through traditional client/server-type interactions, to rich social interactions (the ability to cooperate, coordinate, and negotiate about a course of action).

Whatever the nature of the social process, however, there are two points that qualitatively differentiate agent interactions from those that occur in other software engineering paradigms. First, agent-oriented interactions generally occur through a high-level (declarative) agent communication language (often based on speech act theory [6]). Consequently, interactions are conducted at the knowledge level [5]: in terms of which goals should be followed, at what time and by whom (compare this with method invocation or function calls that operate at a purely syntactic level). Secondly, as agents are flexible problem-solvers, operating in an environment in which they have only partial control and observability, interactions need to be handled in a similarly flexible manner. Thus, agents need the computational apparatus to make context-dependent decisions about the nature and scope of their interactions and to initiate (and respond to) interactions that were not foreseen at design time.

Figure 2. Canonical view of a multiagent system.



Since agents act either on behalf of individuals or companies or as part of some wider initiative, there is typically some underpinning organizational context to agents' interactions. This context defines the nature of the relationship between the agents. For example, they may be peers working together in a team or one may be the manager of the others. To capture such links, agent systems have explicit constructs for modeling organizational relationships (manager, team member). In many cases, these relationships are subject to ongoing change: social interaction means existing relationships evolve (a team of peers may elect a leader) and new relations are created (a number of agents may form a team to deliver a service that no one individual can offer). The temporal extent of these relationships can also vary enormously, ranging from providing a service as a one-off option to a permanent bond. To cope with this variety and dynamics, agent researchers have devised protocols that enable organizational groupings to be formed and disbanded, specified mechanisms to ensure groupings act together in a coherent fashion, and developed structures to characterize the macro behavior of collectives [4, 11].

Drawing these points together (see Figure 2), it can be seen that adopting an agent-oriented approach to software engineering means decomposing the problem into multiple, autonomous components that can act and interact in flexible ways to achieve their set objectives. The key abstraction models that define the agent-oriented mindset are agents, interactions, and organizations. Finally, explicit structures and mechanisms are often used to describe and manage the complex and changing web of organizational relationships that exist between the agents.

The argument in favor of an agent-oriented approach to software engineering includes:

- Show that agent-oriented decompositions are an effective way of partitioning the problem space of a complex system;
- Show that the key abstractions of the agent-oriented mindset are a natural means of modeling complex systems; and
- Show that the agent-oriented philosophy for modeling and managing organizational relationships is appropriate for dealing with the dependencies and interactions that exist in complex systems.

The Merits of Agent-Oriented Decompositions

Complex systems consist of a number of related subsystems organized in a hierarchical fashion (see Figure 1). At any given level, subsystems work together to achieve the functionality of their parent system. Moreover, within a subsystem, the constituent components work together to deliver the overall functionality. Thus, the same basic model of interacting components, working together to achieve particular objectives occurs throughout the system. Given this fact, it is entirely natural to modularize the components in terms of the objectives they achieve.[2] In other words, each component can be thought of as achieving one or more objectives. A second impor-

[2]The view that decompositions based upon functions/actions/processes are more intuitive and easier to produce than those based upon data/objects is even acknowledged within the object-oriented community (see [7]).

tant observation is that complex systems have multiple loci of control: "real systems have no top" [7]. Applying this philosophy to objective-achieving decompositions means the individual components should localize and encapsulate their own control. Thus, entities should have their own thread of control (that is, they should be active) and they should have control over their own actions (that is, they should be autonomous).

For the active and autonomous components to fulfill both their individual and collective objectives, they need to interact (recall complex systems are only nearly decomposable). However the system's inherent complexity means it is impossible to a priori know about all potential links: interactions will occur at unpredictable times, for unpredictable reasons, between unpredictable components. For this reason, it is futile to try and predict or analyze all the possibilities at design time. It is more realistic to endow the components with the ability to make decisions about the nature and scope of their interactions at runtime. From this, it follows that components need the ability to initiate (and respond to) interactions in a flexible manner.

The policy of deferring to runtime decisions about component interactions facilitates the engineering of complex systems in two ways. First, problems associated with the coupling of components are significantly reduced (by dealing with them in a flexible and declarative manner). Components are specifically designed to deal with unanticipated requests and can spontaneously generate requests for assistance if they find themselves in difficulty. Moreover because these interactions are enacted through a high-level agent communication language, coupling becomes a knowledge-level issue. At a stroke this removes syntactic concerns from the types of errors caused by unexpected interactions. Secondly, the problem of managing control relationships between the software components (a task that bedevils traditional objective-based decompositions) is significantly reduced. All agents are continuously active and any coordination or synchronization that is required is handled bottom-up through interagent interaction.

From this discussion, it is apparent that the natural way to modularize a complex system is in terms of multiple autonomous components that can act and interact in flexible ways in order to achieve their set objectives. Given this, the agent-oriented approach is simply the best fit to this ideal.

The Suitability of Agent-Oriented Abstractions

A significant part of the design process is finding the right models for viewing the problem. In general, there will be multiple candidates and the difficult task is picking the most appropriate one. When designing software, the most powerful abstractions are those that minimize the semantic gap between the units of analysis that are intuitively used to conceptualize the problem and the constructs present in the solution paradigm. In the case of complex systems, the problem to be characterized consists of subsystems, subsystem components, interactions and organizational relationships. Taking each in turn:

- Subsystems naturally correspond to agent organizations. They involve a number of constituent components that act and interact according to their role within the larger enterprise.
- The case for viewing subsystem components as agents has been made previously.

The interplay between the subsystems and between their constituent components is most naturally viewed in terms of high-level social interactions: "in a complex system…at any given level of abstraction, we find meaningful collections of objects that collaborate to achieve some higher-level view" [1]. This view accords precisely with the knowledge-level treatment of interaction afforded by the agent-oriented approach. Agent systems are invariably described in terms of "cooperating to achieve common objectives," "coordinating their actions" or "negotiating to resolve conflicts."

Complex systems involve changing webs of relationships between their various components. They also require collections of components to be treated as a single conceptual unit when viewed from a different level of abstraction. Here again the agent-oriented mindset provides suitable abstractions. A rich set of structures are available for explicitly representing organizational relationships. Interaction protocols exist for forming new groupings and disbanding unwanted ones. Finally, structures are available for modeling collectives. The latter point is especially useful in relation to representing subsystems since they are nothing more than a team of components working together to achieve a collective goal.

The Need for Flexible Management of Changing Organizational Structures

Organizational constructs are first-class entities in agent systems—explicit representations are made of organizational relationships and structures. Moreover, agent-oriented systems have the concomitant computational mechanisms for flexibly forming, maintaining, and disbanding organizations. This

representational power enables agent systems to exploit two facets of the nature of complex systems. First, the notion of a primitive component can be varied according to the needs of the observer. Thus at one level, entire subsystems can be viewed as singletons, alternatively teams or collections of agents can be viewed as primitive components, and so on until the system eventually bottoms out. Secondly, such structures provide the stable intermediate forms that are essential for the rapid development of complex systems. Their availability means that individual agents or organizational groupings can be developed in relative isolation and then added into the system in an incremental manner. This, in turn, ensures there is a smooth growth in functionality.

Will Agent-Oriented Techniques Be Widely Adopted?

There are two key pragmatic issues that will determine whether agent-oriented approaches catch on as a software engineering paradigm: the degree to which agents represent a radical departure from current software engineering thinking and the degree to which existing software can be integrated with agents.

A number of trends become evident when examining the evolution of programming models. There has been an inexorable move from languages that have their conceptual basis determined by the underlying machine architecture, to languages that have their key abstractions rooted in the problem domain. Here the agent-oriented world view is perhaps the most natural way of characterizing many types of problems. Just as the real-world is populated with objects that have operations performed on them, so it is equally full of active, purposeful agents that interact to achieve their objectives (see the sidebar for more detailed comparison). Indeed, many object-oriented analyses start from precisely this perspective: "we view the world as a set of autonomous agents that collaborate to perform some higher level function" [1].

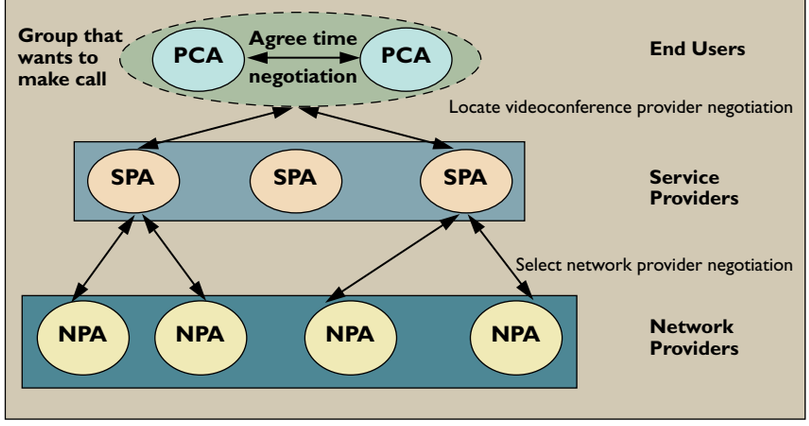
The basic building blocks of the programming models exhibit increasing degrees of localization and encapsulation [8], and agents follow this trend by localizing purpose inside each agent, by giving each agent its own thread of control, and by encapsulating action selection. Additionally, ever-richer mechanisms for promoting reuse are being provided. Here, the agent view also reaches new heights. Rather than stopping at reuse of subsystem components (design patterns and componentware) and rigidly preordained interactions (application frameworks), agents enable whole subsystems and flexible interactions to be reused. In the former case, agent designs and implementations are reused within and between applications. Consider, for example, the class of agent architectures that have beliefs (what the agent knows), desires (what the agent wants) and intentions (what the agent is doing) at its core. Such architectures have been used in a wide variety of applications including air traffic control, process control, fault diagnosis and transportation [4, 8]. In the latter case, flexible patterns of interaction such as the Contract Net Protocol (an agent with a task to complete advertises this

Comparing Object- and Agent-based Approaches

Although there are certain similarities between object- and agent-oriented approaches (both adhere to the principle of information hiding and recognize the importance of interactions), there are also a number of important differences [10]. First, objects are generally passive in nature: they need to be sent a message before they become active. Secondly, although objects encapsulate state and behavior realization, they do not encapsulate behavior activation (action choice). Thus, any object can invoke any publicly accessible method on any other object. Once the method is invoked, the corresponding actions are performed. Additionally, object-orientation fails to provide an adequate set of concepts and mechanisms for modeling complex systems: for such systems "we find that objects, classes, and modules provide an essential yet insufficient means of abstraction" [1]. Individual objects represent too fine a granularity of behavior and method invocation is too primitive a mechanism for describing the types of interactions that take place. Recognition of these facts led to the development of more powerful abstraction mechanisms such as design patterns, application frameworks, and componentware. Although these are undoubtedly a step forward, they fall short of the desired characteristics for complex system development. By their very nature, they focus on generic system functions and the mandated patterns of interaction are rigid and predetermined. Finally, object-oriented approaches provide only minimal support for specifying and managing organizational relationships (basically relationships are defined by static inheritance hierarchies). C

fact to others who it believes are capable of performing it, these agents may submit a bid to perform the task if they are interested, and the originator then delegates the task to the agent that makes the best bid) and various forms of resource-allocation auction (for example, English, Dutch, Vickrey) have been reused in significant numbers of applications. In short, agent-oriented techniques represent a natural progression of current software engineering thinking and, for this reason, the main concepts and tenets of the approach should be readily acceptable to software engineering practitioners.

Figure 3. Dynamic provisioning of virtual private networks by end users.



The second factor in favor of a widespread incorporation of agents is that their adoption does not require a revolution in terms of an organization's software capabilities. Agent-oriented systems are evolutionary and incremental as legacy (non-agent) software can be incorporated in a relatively straightforward manner. The technique used is to place wrapping software around the legacy code. The wrapper presents an agent interface to the other software components. Thus from the outside it looks like any other agent. On the inside, the wrapper performs a two-way translation function: taking external requests from other agents and mapping them into calls in the legacy code, and taking the legacy code's external requests and mapping them into the appropriate set of agent communication commands. This ability to wrap legacy systems means agents may initially be used as an integration technology. However, as new requirements are placed on the system, agents may be developed and added. This feature enables a complex system to grow in an evolutionary fashion (based on stable intermediate forms), while adhering to the important principle that there should always be a working version of the system available.

Case Study: Provisioning a Virtual Private Network

As an exemplar of a complex, distributed system consider the task of dynamically provisioning a public communication network (such as the Internet) as a virtual private network for end users. To be more definitive, let the task in question be setting up a videoconferencing meeting [3]. This application involves a variety of different individuals and organizations (see Figure 3). There are the end users that are each represented by their personal communication agent (PCA). The providers of services on the network (such as setting up a videoconference, for example) are each represented by a service provider agent (SPA). Finally, there are the agents that represent the network provider on whose telecommunications infrastructure the services will actually be delivered (each represented by a network provider agent (NPA)). In setting up a videoconference call, the various PCAs negotiate, on behalf of their particular users, with one another in order to find a suitable time for the call. When they come to an agreement, one of the PCAs then contacts, and subsequently negotiates with, the various SPAs that offer the videoconference service (not all SPAs will do this). This negotiation revolves around the cost of the conference call and the quality of service that is desired. The SPA that wins the contract then negotiates with the various NPAs to determine which of them can deliver the desired quality and bandwidth at the best price.

This application highlights many of the benefits that are have been claimed for an agent-oriented approach to software engineering. Autonomous agents are the most natural means of representing the distinct individuals and organizations that are present in the application. Each such entity is an active problem-solver that has its own objectives to achieve and has control over the actions it chooses and the resources that it expends. The agents need to be responsive to changes in their environment (for example, a NPA may need to arrange additional network capacity from another NPA in order to maintain its agreed upon quality of service if part of its network fails) and they need to be able to opportunistically adopt new goals as they present themselves (for example, two SPAs may discover they have complementary service capabilities and may decide to act together in

order to offer a new service).

A second factor is the agents' need to engage in knowledge-level interactions in order to achieve their individual objectives. In this case, agents typically represent self-interested entities and so the main form of interaction is negotiation. Thus, to set the time of the videoconference or to select a particular service or network provider the agents make proposals, trade offers, make concessions and, hopefully, come to agreements. This rich form of interaction is necessary because the agents represent autonomous stakeholders and also to ensure that agents can arrange their activities in a manner that is appropriate to their prevailing circumstances.

Finally, there is a very clear and explicit notion of organizational context. The application involves a number of different real-world organizations: individual end users, companies that provide the different types of services, and network providers that control the underlying telecommunications infrastructure. These relationships directly affect the agents' behavior. For example, if a SPA and a NPA are in fact part of the same organization, then their negotiations are more cooperative in nature than if they represent two unrelated companies. Similarly, the PCAs that have agreed to hold a conference call act as a team rather than a collection of individuals. Additionally, during the ongoing operation of the application new organizational groupings can appear and then disband. The PCAs of distinct end users form themselves into collectives when they require a particular service (for example, all the participants of the videoconference). Individual SPAs combine their capabilities to offer new services that are beyond the scope of any individual provider. Competing NPAs form themselves into temporary coalitions in order to respond to particularly large requests for network resources.[3]

Conclusion

Agent-oriented techniques are being increasingly used in a range of telecommunication, commercial, and industrial applications. However, if they are to enter the mainstream of software engineering it is vital that clear arguments are advanced as to their suitability for solving large classes of problems (as opposed to specific point solutions). To this end, this article has sought to justify precisely why agent-oriented approaches are appropriate for developing complex, distributed software systems. These general points are then made more concrete by showing how they apply in a specific telecommunications application. In making these arguments, it is possible for proponents of other software engineering paradigms to claim that the key concepts of agent-oriented computing can be reproduced using their technique—this is undoubtedly true. Agent-oriented systems are, after all, computer programs and all programs have the same set of computable functions. However, this misses the point. The value of a paradigm is the mindset and the techniques it provides to software engineers. In this respect, agent-oriented concepts and techniques are both well suited to developing complex, distributed systems and an extension of those currently available in other paradigms. C

[3]In contrast, an object-oriented approach is less suitable for this problem because: it cannot naturally represent the autonomous problem-solving behavior of the constituent components (recall objects do not encapsulate action choice); it has nothing to say about the design of flexible problem-solvers that balance reactive and proactive problem-solving nor about interagent negotiation (other than the fact that it involves message exchanges), and it has no innate mechanism for representing and reasoning with the fact that the agents represent different stakeholder organizations (other than the fact that they are different classes).

References

1. Booch, G. *Object-Oriented Analysis and Design with Applications.* Addison Wesley, 1994.
2. Brooks, F.P. *The Mythical Man-Month.* Addison Wesley, 1995.
3. Faratin, P., Jennings, N.R., Buckle, P. and Sierra, C. Automated negotiation for provisioning virtual private networks using FIPA-compliant agents. In *Proceedings of the 5th International Conference on Practical Application of Intelligent Agents and Multi-Agent Systems.* Manchester, UK, 2000, p. 185–202.
4. Jennings, N.R. and Wooldridge, M., Eds. *Agent Technology: Foundations, Applications and Markets.* Springer Verlag, 1998.
5. Newell, A. The knowledge level. *Artificial Intelligence 18,* 1982, 87–127.
6. Mayfield, J., Labrou, Y., and Finin, T. Evaluating KQML as an agent communication language in M. Wooldridge, J.P. Müller, and M. Tambe, Eds., *Intelligent Agents II*, Springer, 1995, 347–360.
7. Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1988.
8. Parunak, H.V.D. Industrial and practical applications of distributed AI. In G. Weiss, Ed., *Multi-Agent Systems.* MIT Press, 1999, 377-421.
9. Simon, H.A. *The Sciences of the Artificial.* MIT Press, 1996.
10. Wooldridge, M. Agent-based software engineering. In *IEE Proceedings of Software Engineering 144*, 1997, 26–37.
11. Wooldridge, M. and Jennings, N.R. Intelligent agents: Theory and practice. *The Knowledge Engineering Review 10*, 2 (1995), 115–152.

Nicholas R. Jennings (nrj@ecs.soton.ac.uk) is a professor in the Department of Electronics and Computer Science at the University of Southampton, UK.