Agent-Based Computing and Programming of Agent Systems

Michael Luck[1], Peter McBurney[2], and Jorge Gonzalez-Palacios[1]

[1] School of Electronics and Computer Science,
University of Southampton, United Kingdom
mml@ecs.soton.ac.uk, jlgp03r@ecs.soton.ac.uk
[2] Department of Computer Science,
University of Liverpool, United Kingdom
p.j.mcburney@csc.liv.ac.uk

Abstract. The concepts of autonomous agent and multi-agent system provide appropriate levels of abstraction for the design, implementation and simulation of many complex, distributed computational systems, particularly those systems open to external participants. Programming such agent systems presents many difficult challenges, both conceptually and practically, and addressing these challenges will be crucial for the development of agent technologies. We discuss, at a general level, some of the issues involved in programming multi-agent and open, distributed systems, drawing on the recently-published AgentLink III *Roadmap of Agent Based Computing Technologies.*

1 Introduction

Since Shoham's seminal paper [14] on agent-oriented programming in 1993, programming of agent systems has been a key focus of interest. Certainly, the issues surrounding the programming of agent systems had been considered earlier, but Shoham's work marked a key point in identifying agent-oriented programming as a distinct paradigm in a way that had not been done previously. Yet programming agent systems is in fact a broader area that is not restricted to programming paradigms, but includes a whole host of issues ranging from methodological concerns for developers to the specific agent architectures that are required for particular interpreters for agent systems. The purpose of this paper is neither to review the field of programming of agent systems, nor to provide an analysis of particular problems in the area, but more generally to examine the broader context for programming of agents systems in relation to the field of agent-based computing.

As we have argued previously [11], agent technologies can be considered from three perspectives: as a design metaphor; as a source of distinct technologies; and as a simulation tool. Interestingly, the programming of agent systems can be considered in relation to each of these perspectives. First, the programming of agent systems is clearly related to the design metaphor in the development of systems involving large numbers of interacting autonomous components. Second, programming such systems requires the use of particular techniques to develop appropriate architectures and interaction mechanisms. And third, multi-agent systems may also be used as simulations of a system under construction or in operation, thereby assisting developers or controllers with programming of multi-agent systems.

As a design metaphor, agents provide designers and developers with a way of structuring an application around autonomous, communicative elements, and lead to the construction of software tools and infrastructure to support systems development. In this sense, they offer a new and often more appropriate route to the development of complex systems, especially in open and dynamic environments. In order to support this view of systems development, particular tools and techniques need to be introduced. For example, methodologies to guide analysis and design are required; agent architectures are needed for the design of individual components, and supporting infrastructure (including more general, current technologies, such as Web Services) must be integrated.

As a source of technologies, agent-based computing spans a range of specific techniques and algorithms for dealing with interactions with others in dynamic and open environments. These include issues such as balancing reaction and deliberation in individual agent architectures, learning from and about other agents in the environment and user preferences, finding ways to negotiate and cooperate with agents and developing appropriate means of forming and managing coalitions. Moreover, the adoption of agent-based approaches is increasingly influential in other domains. For example, multi-agent systems can provide faster and more effective methods of resource allocation in complex environments, such as the management of utility networks, than any human-centred approach.

As a simulation tool, multi-agent systems can also play a role in programming MAS. One common means to develop a complex, computational system is to proceed through construction of a sequence of prototypes, in a Rapid Applications Development (RAD)-style approach. In such an approach, the successive prototypes effectively act as simulations of the final system under construction; they may be used by the system designers and developers as means to understand system properties under differing conditions or parameter values, and to learn about the system dynamics as agents enter, interact and leave the system. If adequate formal verification methods for multi-agent systems existed, such simulations would not be necessary, but most systems are too complex for the current state-of-the-art in formal verification. For example, the development of online auction systems relies on mathematical game theory for the design of the precise rules of auctions (what economists call *mechanism design*). However, with any more than just a handful of participants, many common auction institutions result in mathematics that is not tractable, and for which analytic solutions may not be known to exist. Thus, auction designers are forced to make assumptions about the participants (for example, that they are utility-maximisers with unbounded processing capabilities) or the mechanism (for example, that communication between participants is instantaneous) that do not apply in the real world. Moreover, designers of *computational* auction systems face problems not envisioned by game theory pioneers, such as malicious participants or participants with buggy code. Accordingly, designers and developers of complex computational systems often build prototypes in which to simulate system performance.[1] Similarly, system controllers may use a simulation of a system in order to diagnose and manage system performance; this is common practice, for example, in utility networks, where a controller may have end-to-end responsibility for quality of service levels provided over physical infrastructures that are not all within the same network. As with the designers' use of MAS systems for simulation, both the end-system being simulated and the system used for simulation are multi-agent systems.

Although this paper does not delve deeply into such distinctions, these are a useful means of drawing out the relevant current and future issues that must be addressed for agent-based computing to see more general application, and for programming and development frameworks to consider if they are to move out of the laboratory, or even provide an alternative to the current *de facto* use of object technologies for building agents systems.

Instead, however, the paper is structured as follows. First, in Section 2, we present in outline form a structure of different levels of abstraction of multi-agent system, which provides a framework through which research and development in agent technologies may be viewed. Then, in Section 3, we discuss agent programming languages and methods, while in Section 4, we consider infrastructure and supporting technologies. Specific aspects of programming open multi-agent systems are then discussed in Section 5, and the paper concludes in Section 6.

2 Levels of Abstraction

There are several distinct high-level trends and drivers, such as Grid computing, ambient intelligence and service-oriented computing, for example, which have led to a heightened interest in agent technologies, and in the low-level computing infrastructures that make them practically feasible. In this context, we consider the key technologies and techniques required to design and implement agent systems that are the focus of current research and development. Because agent technologies are mission-critical for engineering and for managing certain types of information systems, such as Grid systems and systems for ambient intelligence, the technologies and techniques discussed below will be important for many applications, even those not labeled as agent systems. These technologies can be grouped into three categories, according to the scale at which they apply.

– Organisation-level: at the top level are technologies and techniques related to agent societies as a whole. Here, issues of organisational structure, trust, norms and obligations, and self-organisation in open agent societies are

[1] For example, Guala [5] describes the simulation activities — both computational and human — used by economists who advised the US Federal Communications Commission on the development of online mobile spectrum auctions, undertaken since 1994.

paramount. Once again, many of these questions have been studied in other disciplines — for example, in sociology, anthropology and biology. Drawing on this related work, research and development is currently focused on technologies for designing, evolving and managing complex agent societies.
- Interaction-level: these are technologies and techniques that concern communications between agents — for example, technologies related to communication languages, interaction protocols and resource allocation mechanisms. Many of the problems that these technologies aim to solve have been studied in other disciplines, including economics, political science, philosophy and linguistics. Accordingly, research and development is drawing on this prior work to develop computational theories and technologies for agent interaction, communication and decision-making.
- Agent-level: these are technologies and techniques concerned only with individual agents — for example, procedures for agent reasoning and learning. Problems at this level have been the primary focus of artificial intelligence since its inception, aiming to build machines that can reason and operate autonomously in the world. Agent research and development has drawn extensively on this prior work, and most attention in the field of agent-based computing now focuses at the previous two higher levels.

This analysis parallels the abstraction of Zambonelli and Omicini [17] in referring to the macro, meso and micro levels of agent-oriented software engineering, respectively. Within these levels of abstraction, we can consider technologies providing infrastructure and supporting tools for agent systems, such as agent programming languages and software engineering methodologies. These supporting technologies and techniques provide the basis for both the theoretical understanding and the practical implementation of agent systems, and are considered in more detail in the next section. In particular, we outline key issues, and consider likely and important challenges to move forward the technology, in research, development and application.

3 Agent Programming

3.1 Agent Programming Languages

Most research in agent-oriented programming languages is based on declarative approaches, mostly logic based. Imperative languages are in essence inappropriate for expressing the high-level abstractions associated with agent systems design; however, agent-oriented programming languages should (and indeed tend to) allow for easy integration with (legacy) code written in imperative languages. From the technological perspective, the design and development of agent-based languages is also important.

Currently, real agent-oriented languages (such as BDI-style ones) are limited, and used largely for research purposes; apart from some niche applications, they remain unused in practice. However, recent years have seen a significant increase in the degree of maturity of such languages, and major improvements in the development platforms and tools that support them [1]. Current research emphasises the role of multi-agent systems development environments to assist in the development of complex multi-agent systems, new programming principles to model and realise agent features, and formal semantics for agent programming languages to implement specific agent behaviours.

A programming language for multi-agent systems should respect the principle of separation of concerns and provide dedicated programming constructs for implementing individual agents, their organisation, their coordination, and their environment. However, due to the lack of dedicated agent programming languages and development tools (as well as more fundamental concerns relating to the lack of clear semantics for agents, coordination, etc), the construction of multi-agent systems is still a time-consuming and demanding activity.

One key challenge in agent-oriented programming is to define and implement some truly agent-oriented languages that integrate concepts from both declarative and object-oriented programming, to allow the definition of agents in a declarative way, yet supported by serious monitoring and debugging facilities. These languages should be highly efficient, and provide interfaces to existing mainstream languages for easy integration with code and legacy packages. While existing agent languages already address some of these issues, further progress is expected in the short term, but thorough practical experimentation in real-world settings (particularly large-scale systems) will be required before such languages can be adopted by industry, in the medium to long term.

In addition to languages for single agents, we also need languages for high-level programming of multi-agent systems. In particular, the need for expressive, easy-to-use, and efficient languages for coordinating and orchestrating intelligent heterogeneous components is already pressing and, although much research is already being done, the development of an effective programming language for coordinating huge, open, scalable and dynamic multi-agent systems composed of heterogeneous components is a longer term goal.

3.2 Formal Methods

While agent-oriented programming ultimately seeks practical application, the development of appropriate languages demands an associated formal analysis. While the notion of an agent acting autonomously in the world is intuitively simple, formal analysis of systems containing multiple agents is inherently complex. In particular, to understand the properties of systems containing multiple actors, powerful modelling and reasoning techniques are needed to capture possible trajectories of the system. Such techniques are required if agents and agent systems are to be modelled and analysed computationally.

Research in the area of formal models for agent systems attempts to represent and understand properties of the systems through the use of logical formalisms describing both the mental states of individual agents and the possible interactions in the system. The logics used are often logics of belief or other modalities, along with temporal modalities, and such logics require efficient theorem-proving or model-checking algorithms when applied to problems of significant scale.

Recent efforts have used logical formalisms to represent social properties, such as coalitions of agents, preferences and game-type properties.

It is clear that formal techniques such as model checking are needed to test, debug and verify properties of implemented multi-agent systems. Despite progress, there is still a real need to address the issues that arise from differences in agent systems, in relation to the paradigm, the programming languages used, and especially the design of self-organising and emergent behaviour. For the latter, a programming paradigm that supports automated checking of both functional and non-functional system properties may be needed. This would lead to the need to certify agent components for correctness with respect to their specifications. Such a certification could be obtained either by selecting components that have already been verified and validated off-line using traditional techniques such as inspection, testing and model checking or by generating code automatically from specifications. Furthermore, techniques are needed to ensure that the system still executes in an acceptable, or safe, manner during the adaptation process, for example using techniques such as dependency analysis or high level contracts and invariants to monitor system correctness before, during and after adaptation.

4 Infrastructure and Supporting Technologies

Any infrastructure deployed to support the execution of agent applications, such as those found in ambient and ubiquitous computing must, by definition, be long lived and robust. In the context of self-organising systems, this is further complicated, and new approaches supporting the evolution of the infrastructures, and facilitating their upgrade and update at runtime, will be required. Given the potentially vast collection of devices, sensors, and personalised applications for which agent systems and self-organisation may be applicable, this update problem is significantly more complex than so far encountered. More generally, middleware, or platforms for agent interoperability, as well as standards, will be crucial for the medium term development of agent systems.

4.1 Interoperability

At present, the majority of agent applications exist in academic and commercial laboratories, but are not widely available in the real world. The move out of the laboratory is likely to happen over the next ten years, but far greater automation than is currently available in dealing with knowledge management is needed for information agents. In particular, this demands new web standards that enable structural and semantic description of information; and services that make use of these semantic representations for information access at a higher level. The creation of common ontologies, thesauri or knowledge bases plays a central role here, and merits further work on the formal descriptions of information and actions, and, potentially, a reference architecture to support the higher level services mentioned above.

Distributed agent systems that adapt to their environment must both adapt individual agent components and coordinate adaptation across system layers (i.e. application, presentation and middleware) and platforms. In other words interoperability must be maintained across possibly heterogeneous agent components during and after self-organisation actions and outcomes. Furthermore, agent components are likely to come from different vendors and hence the developer may need to integrate different self-organisation mechanisms to meet application requirements. The problem is further complicated by the diversity of self-organisation approaches applicable at different system layers. In many cases, even solutions within the same layer are often not compatible. Consequently, developers need tools and methods to integrate the operation of agent components across the layers of a single system, among multiple computing systems, as well as between different self-organisation frameworks.

4.2 Agent Oriented Software Engineering Methodologies

Despite a number of languages, frameworks, development environments, and platforms that have appeared in the literature, implementing multi-agent systems is still a challenging task. To manage the complexity of multi-agent systems design and implementation, the research community has produced a number of methodologies that aim to structure agent development. However, even if practitioners follow such methodologies during the design phase, there are difficulties in the implementation phase, partly due to the lack of maturity in both methodologies and programming tools. There are also difficulties in implementation due to: a lack of specialised debugging tools; skills needed to move from analysis and design to code; the problems associated with awareness of the specifics of different agent platforms; and in understanding the nature of what is a new and distinct approach to systems development.

In relation to open and dynamic systems, new methodologies for systematically considering self-organisation are required. These methodologies should be able to provide support for all phases of the agent-based software engineering lifecycle, allowing the developer to start from requirements analysis, identify the aspects of the problem that should be addressed using self-organisation and design and implement the self-organisation mechanisms in the behaviour of the agent components. Such methodologies should also encompass techniques for monitoring and controlling the self-organising application or system once deployed.

4.3 Integrated Development Environments

In general, integrated development environment (IDE) support for developing agent systems is rather weak, and existing agent tools do not offer the same level of usability as state-of-the-art object oriented IDEs. One main reason for this is the previous unavoidable tight coupling of agent IDEs and agent platforms, which results from the variety of agent models, platforms and programming languages.

This is now changing, however, with an increased trend towards modelling rather than programming.

With existing tools, multi-agent systems often generate a huge amount of information related to the internal state of agents, messages sent and actions taken, but there are not yet adequate methods for managing this information in the context of the development process. This has impacts both for dealing with the information generated in the system and for obtaining this information without altering the design of the agents within it.

Platforms like JADE provide general introspection facilities for the state of agents and for messages, but they enforce a concrete agent architecture that may not be appropriate for all applications. Thus, tools for inspecting any agent architecture, analogous to the remote debugging tools in current object-oriented IDEs, are needed, and some are now starting to appear [2]. Extending this to address other issues related to debugging for organisational features, and for considering issues arising from emergence in self-organising systems will also be important in the longer term. The challenge is relevant now, but will grow in importance as the complexity of installed systems increases further.

The inherent complexity of agent applications also demands a new generation of CASE tools to assist application designers in harnessing the large amount of information involved. This requires providing reasoning at appropriate levels of abstraction, automating the design and implementation process as much as possible, and allowing for the calibration of deployed multi-agent systems by simulation and run-time verification and control.

More generally, there is a need to integrate existing tools into IDEs rather than starting from scratch. At present there are many research tools, but little that integrates with generic development environments, such as Eclipse; such advances would boost agent development and reduce implementation costs. Indeed, developing multi-agent systems currently involves higher costs than using conventional paradigms due to the lack of supporting methods and tools.

The next generation of computing system is likely to demand large numbers of interacting components, be they services, agents or otherwise. Current tools work well with limited numbers of agents, but are generally not yet suitable for the development of large-scale (and efficient) agent systems, nor do they offer development, management or monitoring facilities able to deal with large amounts of information or tune the behaviour of the system in such cases.

4.4 Metrics

Metrics for agent-oriented software are also needed: engineering always implies some activity of measurement, and traditional software engineering already uses widely applied measuring methods to quantify aspects of software such as complexity, robustness and mean time between failures. However, the dynamic nature of agent systems, and the generally non-deterministic behaviour of self-organising agent applications deem traditional techniques for measurement and evaluation inappropriate. Consequently, new measures and techniques for both quantitatively and qualitatively assessing and classifying multi-agent systems applications (be they self-organising or not) are needed.

5 Open Systems

5.1 Introduction

The advent of low-cost computation has driven the proliferation of computers and computational devices over the last two decades, and with them, a proliferation of physical networks. With the interconnections provided by physical networks come requirements for computational societies — means of distributing data gathering, processing intelligence, resource allocation and system control, as in ambient intelligence, pervasive computing, Grid computing, etc. These computational societies increasingly require applications which are large-scale, decentralised, proactive, situated and open.

However, traditional approaches (for example, object-oriented and component-based computing) have fallen short in engineering these types of application because they utilise too low a level of abstraction, focusing on the *"physical distribution of data, resources and processes,"* rather than on the *"logical distribution of responsibility, control and regulation,"* to quote Pitt's apt distinction [13, p. 140]. As a consequence, alternative approaches have been proposed, with a consensus emerging that the concept of autonomous agency provides the appropriate level of abstraction to successfully develop these types of systems [8]. As mentioned above, this has led to the appearance of several agent-oriented software methodologies, which aim to support development of open systems.

More specifically, an open system is a one that allows run-time incorporation of components that may not be known at design time. In addition, the components of an open system are typically not designed and developed by the same design team or for the same stakeholders, and different teams may use different development tools or adopt different policies or objectives, leading to the appearance of self-interested components. In whatever way or for whom the components are developed, they will normally have the right to access the corresponding facilities provided by the system, as well as having an obligation to adhere to the system's rules.

5.2 Specification of Open Systems

Even though several methodologies exist to support the development of open systems, these present drawbacks when dealing with the incorporation of new components to an existing system. In fact, the phase of operation of a system is generally not considered in methodologies. For example, despite the importance of providing developers with a description of system properties, the problem of *what* must be present in such a description and *how* it must be described has been scarcely addressed.

Although each system can have its own form of specification (comprising a list of the features provided by the system, as well as the requirements to join it), a common *model* of specification would bring multiple benefits. First, by using predefined models, the time spent on creating specifications can be reduced. Second, using common models would promote the creation of standards, of which much has already been written, and much work undertaken. Finally, by defining and structuring the description of a system, the model of specification is valuable in defining the inputs of run-time components that check system properties, for example that new agents comply with required characteristics, and do not violate operating principles.

In particular, because of the autonomous and pro-active nature of the components (agents) of multi-agent systems, such specifications are likely to be vital in appropriately enabling and regulating self-interested behaviour in open system. This suggests that tools and techniques for the development of open systems (including the above discussed methodologies, design tools, monitoring tools, debuggers, platforms, and so on) will be required if the requirements of large-scale open systems that underlie visions of Grid computing and similar paradigms are to be satisfied. In addition, a focus on standardisation of abstractions, operating models, interaction protocols and other *patterns* of activity will be needed.

In relation to these issues, we outline below some further concerns in the context of the current situation that require further research and development.

5.3 Methodologies

Current methodologies are not complete or detailed enough. Even if we restrict our attention only to the analysis and design phases of the development cycle, very few methodologies cover all the corresponding activities. For example, SODA does not address intra-agent issues, Gaia does not consider how the services an agent provides are realised [15], Gaia extended with organisational abstractions is not sufficiently elaborated [16], the models of MAS-CommonKADS require further improvement, as recognised by its developers [7], and the evaluation of MESSAGE reports that it is not complete nor mature [9].

5.4 Agent Architectures

Despite the focus on development and programming, it is impossible not to consider specific agent architectures during the design and programming of individual agents. Different approaches are taken by different methodologies, with three significant styles as follows.

- For some methodologies, architectures are outside their scope since they were designed only for analysis and high-level design, for example, Gaia [15].
- Some methodologies and languages are tied to a specific architecture. For example the methodological work of Kinny, Georgeff and Rao [10] is tied to the BDI architecture.
- Some other methodologies are tied to a specific architecture but it is suggested that the same principles can be applied to other architectures, or that other architectures can be adapted to fit. For example, MESSAGE [9] uses the generic Layer Architecture for this purpose.

While there are clearly efforts to incorporate architectures into methodologies, no methodology satisfactorily incorporates even a subset of the most popular agent architectures. Conversely, the development of programming languages seems to require commitment to particular architectures (see, for example, 3APL [6], AgentSpeak(L) [3], and AGENT-0[14]), yet mapping from one to another is problematic. The variety of domains of application of agent-based computing, especially in open systems, requires programming languages such as these to be broadly applicable; for this to be the case, some solution, technical or consensual, is needed.

5.5 Interactions in Open Systems

Interactions are a key element in both the modelling and operation of multi-agent systems, since agents achieve their goals by interacting with other agents. In the case of open systems, interaction mechanisms should be flexible enough to allow new agents to be incorporated into the system. However, most methodologies neither facilitate nor obstruct the development of open systems, and some effort is required to explicitly consider this issue. For example, enriching the analysis and upper design phases of MESSAGE [9] with the organisational abstractions recommended by Zambonelli *et al.* [16] could lead to a methodology suitable for tackling open systems.

6 Conclusions

In any high-technology domain, the systems deployed in commercial or industrial applications tend to embody research findings somewhat behind the leading edge of academic and industrial research. Multi-agent systems are no exception to this, with currently-deployed systems having features found in published research and prototypes of three to five years ago. By looking at current research interests and areas of focus, we were therefore able in [12] to extrapolate future trends in deployed systems, which we classified into four broad phases of development of multi-agent system technology over the next decade; we summarise these phases here.

The four phases are, of necessity, only indicative, since there will always be some companies and organisations which are leading users of agent technologies, pushing applications ahead of these phases, while many other organisations will not be as advanced in their use of the technology. We aim to describe the majority of research challenges at each time period.

6.1 Phase 1: Current

Multi-agent systems are currently typically designed by one design team for one corporate environment, with participating agents sharing common high-level goals in a single domain. These systems may be characterised as closed. (Of course, there is also work on individual competitive agents for automated negotiation, trading agents, and so forth, but typically also constrained by closed environments.) The communication languages and interaction protocols are typically in-house protocols, defined by the design team prior to any agent interactions. Systems are usually only scalable under controlled, or simulated, conditions. Design approaches, as well as development platforms, tend to be ad hoc, inspired by the agent paradigm rather than using principled methodologies, tools or languages. Although this is still largely true, there is now an increased focus on, for example, taking methodologies out of the laboratory and into development environments, with commercial work being done on establishing industrial-strength development techniques and notations. As part of this effort, some platforms now come with their own protocol libraries and force the use of standardised messages, taking one step towards the short-term agenda.

6.2 Phase 2: Short-Term Future

In the next phase of development, systems will increasingly be designed to cross corporate boundaries, so that the participating agents have fewer goals in common, although their interactions will still concern a common domain, and the agents will be designed by the same team, and will share common domain knowledge. Increasingly, standard agent communication languages, such as FIPA ACL [4], will be used, but interaction protocols will be mixed between standard and non-standard ones. These systems will be able to handle large numbers of agents in pre-determined environments, such as those of Grid applications. Development methodologies, languages and tools will have reached a degree of maturity, and systems will be designed on top of standard infrastructures such as web services or Grid services, for example.

6.3 Phase 3: Medium-Term Future

In the third phase, multi-agent systems will permit participation by heterogeneous agents, designed by different designers or teams. Any agent will be able to participate in these systems, provided their (observable) behaviour conforms to publicly-stated requirements and standards. However, these open systems will typically be specific to particular application domains, such as B2B eCommerce or bioinformatics. The languages and protocols used in these systems will be agreed and standardised, perhaps drawn from public libraries of alternative protocols that will, nevertheless, likely differ by domain. In particular, it will be important for agents and systems to master this semantic heterogeneity. Supporting this will be the increased use of new, commonly agreed modelling languages (such as Agent-UML, an extension of UML 2.0), which will promote the use of IDEs and, hopefully, start a harmonisation process as was the case for objects with UML.

Systems will scale to large numbers of participants, although typically only within the domains concerned, and with particular techniques (such as domain-bridging agents), to translate between separate domains. System development will proceed by standard agent-specific methodologies, including templates and patterns for different types of agents and organisations. Agent-specific programming languages and tools will be increasingly used, making the use of formal verification techniques possible to some extent. Semantic issues related to, for example, coordination between heterogeneous agents, access control and trust, are of particular importance here. Also, because these systems will typically be open, issues such as robustness against malicious or faulty agents, and finding an appropriate trade-off between system adaptability and system predictability, will become increasingly important.

6.4 Phase 4: Long-Term Future

The fourth phase in this projected future will see the development of open multi-agent systems spanning multiple application domains, and involving heterogeneous participants developed by diverse design teams. Agents seeking to participate in these systems will be able to learn the appropriate behaviour for participation in the course of interacting, rather than having to prove adherence before entry. Selection of communications protocols and mechanisms, and of participant strategies, will be undertaken automatically, without human intervention. Similarly, *ad hoc* coalitions of agents will be formed, managed and dissolved automatically. Although standard communication languages and interaction protocols will have been available for some time, systems in this phase will enable these mechanisms to emerge by evolutionary means from actual participant interactions, rather than being imposed at design time. In addition, agents will be able to form and re-form dynamic coalitions and virtual organisations on-the-fly and pursue ever-changing goals through appropriate interaction mechanisms for distributed cognition and joint action. In these environments, emergent phenomena will likely appear, with systems having properties (both good and bad) not imagined by the initial design team. Multi-agent systems will be able, adaptable and adept in the face of such dynamic, indeed turbulent, environments, and they will exhibit many of the self-aware characteristics described in the autonomic computing vision.

By this phase, systems will be fully scalable in the sense that they will not be restricted to arbitrary limits (on agents, users, interaction mechanisms, agent relationships, complexity, etc). As previously, systems development will proceed by use of rigorous agent-specification design methodologies, in conjunction with programming and verification techniques.

6.5 Realization

Of course, achieving the ambitious future vision outlined in these four phases of development will not proceed without obstacles. Significant research, development and deployment challenges exist for both academic researchers and for commercial developers of agent computing technologies. Languages and methodologies for programming autonomous agents and multi-agent systems are among the most important of these, and will remain at the centre of agent-based computing for at least the next decade.

Acknowledgements

This paper is in part based on, and borrows heavily from, the AgentLink III Roadmap [12] which, in turn, drew on inputs and contributions from the PROMAS Technical Forum Group (as well as others) during the lifetime of AgentLink III.

References

1. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications.* Springer, 2005.
2. J. Botia, A. Lopez-Acosta, and A. Gomez-Skarmeta. Aclanalyser: A tool for debugging multi-agent systems. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence*, pages 967–968, 2004.
3. M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):233–260, 1998.
4. FIPA. Communicative Act Library Specification. Standard SC00037J, Foundation for Intelligent Physical Agents, 3 December 2002.
5. F. Guala. Building economic machines: The FCC Auctions. *Studies in the History and Philosophy of Science*, 32(3):453–477, 2001.
6. K. V. Hindriks, F. S. de Boer, W. van der Hoek, , and J-J. Ch. Meyer. Formal semantics for an abstract agent programming language. In *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, Lecture Notes in Artificial Intelligence, Volume 1365, pages 215–229. Springer-Verlag, 1998.
7. C. A. Iglesias, M. Garijo, J. C. Gonzalez, and J. R. Velasco. Analysis and design of multiagent systems using mas-commonkads. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages*, Lecture Notes in Artificial Intelligence, Volume 1365, pages 313–326. Springer-Verlag, 1998.
8. N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.
9. P. Kearney, J. Stark, G. Caire, F. J. Garijo, J. J. Gomez Sanz, J. Pavon, F. Leal, P. Chainho, and P. Massonet. Message: Methodology for engineering systems of software agents. Technical Report EDIN 0223-0907, Eurescom, 2001.
10. D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of bdi agents. In W. van der Velde and J. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Lecture Notes in Artificial Intelligence, Volume 1038, pages 56–71. Springer-Verlag, 1996.
11. M. Luck, P. McBurney, and C. Preist. A manifesto for agent technology: Towards next generation computing. *Autonomous Agents and Multi-Agent Systems*, 9(3):203–252, 2004.
12. M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction. A Roadmap for Agent Based Computing.* AgentLink III, 2005.

13. J. Pitt. The open agent society as a platform for the user-friendly information society. *AI and Society*, 19:123–158, 2005.
14. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
15. M. Wooldridge, N. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.
16. F. Zambonelli, N. R. Jennings, and M. Wooldridge. Organisational rules as an abstraction for the analysis and design of multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):303–328, 2001.
17. F. Zambonelli and A. Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, 2004.