

AgentPrIME: Adapting MAS Designs to Build Confidence

Simon Miles¹, Paul Groth², Steve Munroe², Michael Luck¹, Luc Moreau²

¹ Department of Computer Science
King's College London
London, WC2R 2LS, UK

² School of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK

Abstract. The products of systems cannot always be judged at face value: the process by which they were obtained is also important. For instance, the rigour of a scientific experiment, the ethics with which an item was manufactured and the use of services with particular licensing all affect how the results of those processes are valued. However, in systems of autonomous agents, and particularly those with multiple independent contributory organisations, the ability of agents to choose how their goals or responsibilities are achieved can hide such *process qualities* from users. The issue of ensuring that users are able to check these process qualities is a software engineering one: the developer must decide to ensure that adequate data is recorded regarding processes and safeguards implemented to ensure accuracy. In this paper, we describe AgentPrIME, an adjunct to existing agent-oriented methodologies that allows system designs to be adapted to give users confidence in the results they produce. It does this by adaptations to the design for *documentation, corroboration, independent storage* and *accountability*.

1 Introduction

Agent-based systems have particular qualities that require their activity to be justified to their users. First, the basis of such systems on autonomous components can mean that decisions which make use of expert knowledge or have significant consequences are handled by software, and so the decisions made by such software must be seen to be reliable if it is to be widely adopted. Also, by having multiple, distributed points of control, an application may rely on services not under the authority of the user, and whose side-effects may not be made apparent to the user: a user may wish to know that the services do not produce their results in an undesirable way, e.g. illegal, unethical etc. Finally, in systems where agents represent localised concerns of distributed users, it is important to know that agents have not released private information more widely than desired.

At some level, this problem has been well researched. Approaches exist to formally specify a multi-agent system, enabling developers to verify its desirable

properties. However, this does not in itself inform developers about what factors need to be considered, nor is it (commercially) realistic to assume fine-grained knowledge of third-party services used in an application. Mechanisms have been designed to guide agents behaviour to reliable results or to constrain agent behaviour to only desirable results, including contracts, norms, protocols, trust evaluations etc.

We argue that, even with this breadth of beneficial technology, there are significant outstanding issues. First, agent-based systems must be designed not just to be reliable but to make their reliability apparent to users, if they are to have *confidence* in the system. Second, the above mechanisms concentrate on the value or otherwise of *results* or the *cost* of achieving those results, both aspects of the system which can be immediately judged by the user or an agent acting on their behalf. Because of this emphasis, other, hidden but still important, aspects are ignored. In particular, the mechanisms do not address how to determine *process qualities* which are not immediately apparent in the result returned by an agent but have an impact on its worth. Examples of important process qualities occur in many domains, such as the following.

- The rigour of the scientific experiment which produced some result.
- The ethics (fair trade, environmental impact etc.) of the process that led to the sale of an item.
- The use of services with licenses which make a result unpatentable.
- The actual inter-dependence of two apparently independent recommendations.

The qualities of the process that led to a result are all evident in the *provenance* of that result, i.e. everything that caused the result to be as it is. For the provenance of a result, and process qualities evident from it, to be made apparent to a user requires that the agent-based system be engineered to record or gain access to adequate information to determine both (1) what has occurred in the system prior to the result being produced, and (2) which of those events are causally related to the eventual result.

However, in a system of flexible autonomous agents, such agents may lie or collude to hide what actions they have taken where it is in their interests to do so (as is true in many of the process examples above). Without specifically designing a system to mitigate for agents' inaccuracy, a user can again be misled. Therefore, we argue that agent-oriented designs must be specifically adapted to mitigate for inaccuracy and provide confidence that users can find out exactly how a multi-agent system came to produce a result.

In this paper, we describe an approach, *AgentPrIME*, to determining what information needs recording, how to adapt the relevant agents to do so, and what must be established of an agent owned by a third-party in order to rely on it to provide compatible and verifiable information regarding provenance. It is not intended itself to be a methodology for the design of agent-based systems, as this is already well covered elsewhere, but is a *methodology fragment*, applied as an adjunct to such methodologies, which increases the reliability of the systems designed.

2 AgentPrIME

A *methodology fragment* [9] is an software engineering procedure that is used in addition to other stages of a methodology in designing an application. It aims to add or ensure some functionality of the system which may otherwise not be guaranteed by the original methodology. Aspect-oriented software engineering [7] provides an example of methodology fragments that provide functionality pervading across a design (usually object-oriented). Others have applied aspect-orientation to agent-based systems [3], but we do not use the aspect concept here, because, while it is not entirely inappropriate, it carries connotations of cutting across agents in a way that pre-supposes that the process they are involved in is fixed at design-time. Process qualities regard processes that have already occurred in a system that may be flexible, open and unreliable.

A desirable quality of a methodology fragment is *methodology neutrality*, in being general and well-defined enough to be applied as part of as many methodologies as possible. This is a distinct quality from the comparable requirement of methodologies (and their fragments) being *widely applicable* to a range of applications.

AgentPrIME is a methodology fragment for agent-oriented software engineering methodologies. We will refer to the methodology to which it is acting as an adjunct as the *extended methodology*. The result of AgentPrIME is a set of *adaptations* to be applied to a system design, so that queries regarding provenance can be reliably answered. It builds on an existing methodology fragment, Provenance Incorporation Methodology (PrIME), described elsewhere [11], which considers adapting software to help users determine provenance of results, but looks only at service-oriented systems and does not address issues relevant to an agent-oriented design, where autonomous components choose their own methods to achieving their goals and so may be dishonest.

There are two aims of AgentPrIME: (1) to make the provenance of results available to users of the system, and (2) to ensure that, as far as possible, the provenance is accurate even when agents in the system may be unreliable. Specifically, AgentPrIME follows has two phases, described in detail in the following sections.

- Identify the causes of agent actions in the design, instances of which will be recorded as the agents act. This phase results in adaptations to agents so that they record such causes for users to later query.
- Identify where additional guarantees of accuracy are required, so as to be able to rely on what agents have recorded. This phase results in adaptations to the interactions between agents, so that users can have more confidence that what agents have recorded is accurate.

AgentPrIME relies on understanding what type of agents will exist within a system, so that their effects in processes can be understood, and the interactions possible between those agents. It can affect both how those agents are ultimately implemented, and may alter the possible interactions between them, as will be seen in the subsequent sections. These dependencies mean that AgentPrIME is

ideally applied at a particular point in the extended methodology, when the design is sufficiently well developed to adapt but not so far developed that effort is wasted. To be more concrete, we define below at what point AgentPrIME would apply when using various methodologies that the reader may be familiar with.

- In Gaia [15], AgentPrIME must be applied after the *agent model* and *acquaintance model* have been completed. This is because it applies to *agent types*, where the functionality of an agent of each type is well-defined, and the interactions between them dictated by the acquaintance model.
- In MaSE [2], AgentPrIME must be applied after the *agent classes* and *conversations* have been created, for an analogous reasons to those given above for Gaia.
- In Prometheus [13], AgentPrIME operates on the *agent overview*, so after the architecture design and before the detailed design.
- In SODA [12], AgentPrIME requires the data from the *interaction* and *agent models*, so applies after SODA has been fully applied.

3 Causality in Multi-Agent Systems

In this section, we describe the basis of AgentPrIME on documenting causal relationships between agent actions. This gives users the facilities to determine the provenance of agent’s actions and outputs. We will consider the unreliability of agents in the next section.

3.1 Causality within Agents

A key part of AgentPrIME is to allow agents to document the *causes* of their actions, so that this information can later be used to determine what occurred in a process. The possible causes in a model depend on the extended methodology, but we will discuss some examples in this section and then show how these can be generalised in a well-defined way for methodology neutrality.

A variety of factors influence an agent’s behaviour at a given instant, and examples are summarised in Figure 1. Here, we are concerned with behaviour that affects the environment, i.e. actions, shown as the *output* of the agent. The influencing factors, depending on the agent model used by a methodology, can include the agent’s *goals*, its *responsibilities* or *rights*. Often the latter factors are due to the *role(s)* that an agent is playing within a system at that instant, the goals etc. having been allocated to the roles in applying the extended methodology [8]. Additionally, triggers from the environment, in which we include messages from other agents, will influence how an agent acts, shown as *input* in the figure.

AgentPrIME, and its supporting technologies, allow an agent to assert the *causal relationship* between two *occurrences*. These assertions, called *relationship p-assertions*, can be stored for later interrogation by a user, as discussed further below. Applied to an agent design, this means that relationships can be asserted

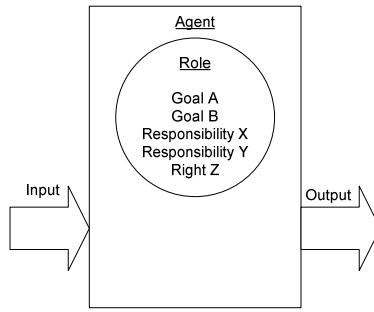


Fig. 1. Potential causes of an agent's actions

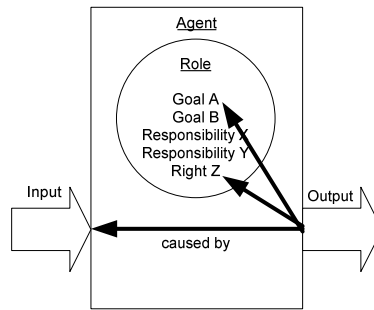


Fig. 2. Causal relationships between an agent's action and its causes

between an output (the effect) and the inputs, goals, responsibilities etc. that caused it to take place. These relationships are depicted in Figure 2. Using the examples already described, these causal relationships can be used to assert:

- That an output message was sent in response to an input message.
- That an action was taken to attempt to fulfil a goal.
- That an action was taken because it was part of the responsibilities of the agent.
- That an action was taken because it was allowed for by a right of the agent.

In recent work [10], we have discussed one particular example of this: how the documentation of the causal effects of goals can be used to make applications more robust (that work focuses on answering particular questions once causal and intentional information is present, rather than the methodological issues covered here).

However, the concepts described above are only a subset of those used in agent-oriented methodologies. Others include motivations, beliefs, intentions, adherence to protocols etc. Many of these may be asserted as causes of an agent’s action. In order for AgentPrIME to be methodology neutral, we need a general definition of whether something specified as part of applying a methodology is a causal relationship. We take the following definition derived from work in the philosophy of mind [6].

E was caused by *C*, if *E* would not have occurred without *C* not having occurred, all else being equal.

By applying this definition, we can determine whether a particular factor influenced an action regardless of methodology. For example, we can say that a particular action would not have been taken if the agent didn’t have a responsibility to do so, or that an action would not have occurred (because it could not) if the agent did not have the right to do so. The important quality of this definition is that it is *system independent*, relying only on a notion of occurrence.

We note that this specification of cause and effect relies on the agents being relatively deterministic at design time, rather than adaptive. While our general technology does not rely on this (agents can assert the causal relationships they are aware of regardless of what was known at design time), the methodology fragment presented here only applies to agents whose behaviour is determined at design-time. Recording the effects on the internal contents of agents, e.g. learning, is also possible using our approach but is out of scope of this paper. Our solution to the latter can be briefly summarised as follows: if a developer wishes to assert effects on the internals of an agent, they can do this by decomposing the agent (for the purposes of the model) and showing how particular internal components, e.g. the agent’s knowledge base, are effected by external stimuli. This approach allows arbitrary complexity and a consistent causal model. Further details on describing these different granularity processes are published elsewhere [11].

3.2 Causality between Agents

One of the causes of an agent's actions discussed above is a message received from another agent. This is of particular interest when examining process qualities: it is not the actions of a single agent that matter but of a set of agents that ultimately produce some result. Therefore, in addition to asserting causal relationships, AgentPrIME allows agents to assert the inputs it receives and outputs it sends to other agents. These assertions are called *interaction p-assertions*, and, along with relationship p-assertions, connect together the actions of one agent to those of another.

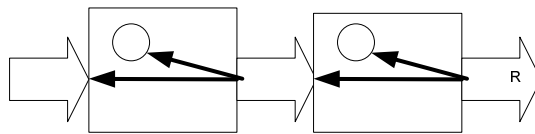


Fig. 3. A chain of interactions and causal relationships between agents

A chain of two agents is shown in Figure 3. In this figure the left-hand agent sends a message to the right-hand agent. This behaviour by the left-hand agent is caused by the factors discussed in the previous section, possibly including communications from other agents. The right-hand agent may act on the basis of receiving the message, and this may include sending messages to other agents. Thus, an adequate collection of interaction and relationship p-assertions provides a connected trail of the process that led to a result. From the result, R, shown in the figure we can follow the causal relationships and interactions back to determine all the factors that ultimately caused it to be as it is. Note that here, we are describing the actual interactions that an agent will have at run-time, assuming that by design and by context it has chosen to interact in that way. How to allocate interactions to agents to best meet system requirements has been addressed by others [1].

3.3 The Wrapper Adaptation

The p-assertions described above must be recorded into repositories so that users can later query them. We call such repositories *provenance stores*. The way that the recording of interaction and relationship p-assertions can be realised in a system, is to apply a *wrapper* to each agent that is to record. The conception of a wrapper is shown in Figure 4. As messages come into or leave an agent, the wrapper records interaction p-assertions regarding their content and relationship p-assertions regarding their causes. Generic wrappers are reusable, allowing adapting agents implemented using a particular technology to be wrapped regardless of the application, e.g. we have implemented generic wrappers for agents that exchange SOAP messages using the Java Axis engine, and shell scripts to adapt agents implemented as UNIX command-line executables [5].

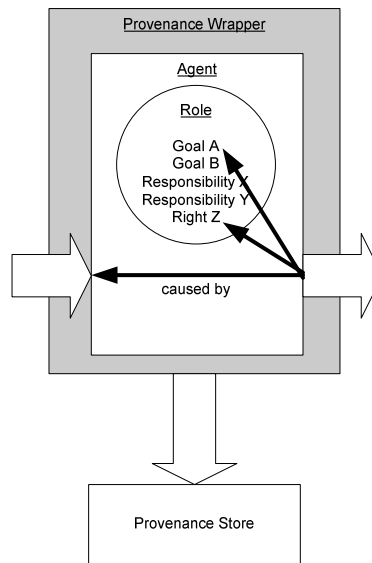


Fig. 4. Wrappers are adaptations to agents that automatically document incoming and outgoing messages, and causal relationships, and send them to a provenance store

3.4 Provenance

An important part of our approach is to use a common, open data model for p-assertions. This means that all agents can independently and autonomously record documentation of their activities in the same format, and a user can examine and interpret this documentation without relying on those agents. The full data model is documented elsewhere [4].

By examining the provenance of a result, we can determine the procedure that was followed to produce it. In theory, this would allow us to check the rigour of a scientific result, whether businesses with dubious ethical records were used in manufacturing a good etc. However, this depends on those agents involved in a process to accurately document what they do, an unreasonable assumption in many domains to which this research would provide most benefit. In the following section, we show how AgentPrIME tackles the problem of potentially dishonest agents.

4 Designing for Accuracy

In this section, we discuss in what ways agents' inaccuracy can obscure process qualities, and how AgentPrIME mitigates these problems through *third-party storage*, *accountability* and *corroboration*. It should be emphasised that these solutions do not *guarantee* accurate, honest documentation, but merely reduce the possibilities for deception.

4.1 Design Levels

Mitigating for inaccuracy can be expensive, and not every application of AgentPrIME needs to incur all of this expense. For instance, a multi-agent system may be completely trusted to not maliciously produce incorrect assertions, e.g. if all agents are owned by a single trusted organisation, but still may do so through error. It is important, therefore, that AgentPrIME allows developers to apply the degree of mitigation they consider most appropriate for a given application.

We classify the types of application, and the design requirements due to them, into three levels, increasing in development cost, as shown in Table 1. A *reliable system* is one in which the agents are assumed to always record complete and accurate documentation, or at least complete and accurate enough that any mitigation would be more costly than it is worth. A *transparent system* is one in which the agents cannot always be trusted to assert correct information but for which there exist ways to corroborate what they have asserted. An *exploitable system* is one in which some agents are free to withhold information about their activities or give false information without being detectable. The latter two types of system will be characterised more concretely in the following section.

System Type	Design Type
Reliable	Design for verification
Transparent	Design for accountability
Exploitable	Design for corroboration

Table 1. Levels of application risk and the design approaches to match them

Reliable systems require no mitigation for inaccuracy, and so applying AgentPrIME to such a system means that agents are simply adapted to record the p-assertions needed to verify what has occurred: *design for verification*. Transparent and exploitable systems require design adaptations to ensure that agents can be held *accountable* for their documentation and that the documentation can be *corroborated*. AgentPrIME tackles this through a series of techniques.

It is important to note that the systems that need to be adapted to mitigate inaccuracy are exactly those systems that users may suspect of recording inaccurate documentation. The incentive for the designers of such systems to apply the adaptations is that users can check whether they have been applied and will trust the results produced by such systems on that basis. That is, regardless of whether a system is reliable or not, a user can choose to trust results from that system only if it is both *(i)* clear from a result’s provenance that it was produced in a legitimate way, and *(ii)* clear from the provenance and other system components described below that the designs were adequate to prevent inaccuracy. AgentPrIME, therefore, provides benefits to two parties:

- For the user, it provides a way to check that adequate safeguards were in place to ensure the provenance is reliable.

- For the system designer, it provides a way to give the necessary guarantees of accuracy to a user.

4.2 Corroboration

We now characterise the difference between transparent and exploitable unreliable systems, and show how AgentPrIME requires more adaptations to be applied to the latter.

Returning to the causal chain shown in Figure 3, we note that for every message in the system, two agents are involved: the *sender* and the *receiver*. If both agents record interaction p-assertions documenting the fact and content of the message they sent/received, then one agent’s assertion can be used to verify the correctness of the other’s assertion. We say that each agent’s *view* of the interaction provides *corroboration* of the other view. Therefore, where an interaction involves one reliable agent and one unreliable agent, the latter’s view of what occurred can be checked. Note, that this cannot apply to the causal relationships: only an agent knows whether its actions were caused by a particular goal, responsibility etc.

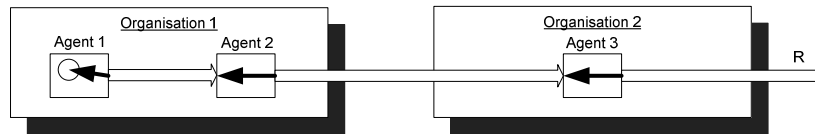


Fig. 5. Multiple organisations involved in a process

The kinds of process that cause most problems are those that involve multiple *organisations*, where each organisation owns a set of agents involved in the process. This is problematic because one organisation can provide an honest facade for another, e.g. an apparently reliable shop may use an ethical supplier. We depict such a scenario in Figure 5, in which Agent 3 in Organisation 2 produces result R partly on the basis of the operations of Agents 1 and 2 in Organisation 1.

Organisations provide a unit of trust: agents can be grouped into organisations such that all agents in an organisation are trusted independently from those in any other organisation. If, in the process shown in Figure 5, Organisation 1 is trusted, then the system as a whole can be said to be transparent. This is because every agent is either trusted or, if not, every interaction they have in a process is with a trusted agent and can therefore be corroborated by examining the p-assertions of the trusted agent.

An alternative situation arises when Organisation 2 is trusted, but Organisation 1 is not. In this case, one of the agents’ assertions cannot be corroborated. The situation, from a user’s point of view is shown in Figure 6: only Agents

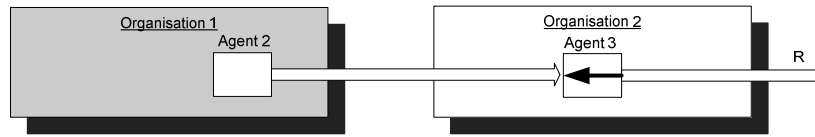


Fig. 6. Opaque organisations involve actions that cannot be verified

2 and 3 produce p-assertions that can be relied on. We say that, in this case, Organisation 1 is *opaque* because part of its process, possibly a significant part from the user’s perspective, is not reliably documented.

4.3 The Corroboration Adaptation

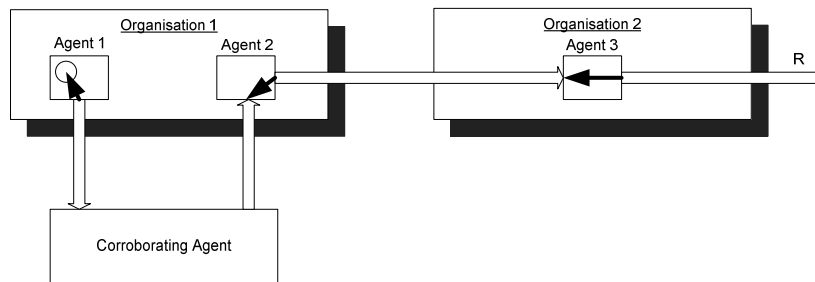


Fig. 7. Corroborating agents can be introduced to ensure a process remains transparent

Applying AgentPrIME to the case above, means that additional agent interactions are introduced to the application process. As shown in Figure 7, the *corroborating agent* is introduced into the process, so that instead of a direct interaction between Agents 1 and 2, this corroborating agent acts as a redirecting intermediary. The corroborating agent must record its own documentation of its interactions, and be trusted by a user to be of value.

4.4 The Third-Party Storage Adaptation

The second technique to mitigate inaccuracy is for agents to store documentation in third-party provenance stores, trusted by both the system owner and the user. These repositories should ensure immutability and longevity of the assertions they contain. Therefore, users have assurance that if accurate data is recorded, it cannot later be altered or deleted. Provenance stores independent from the agents recording documentation is recommended for all types of system, even reliable ones. We assume that the provenance stores used have adequate

transport-level authentication, and access control to prevent malicious editing of data.

4.5 The Accountability Adaptation

The third technique is to ensure that it is possible to verify the origin of every p-assertion recorded, i.e. which agent created it. This is important for every type of unreliable system, including those that are unreliable through error rather than malice, as it allows the faulty agents to be pinpointed within a system. Accountability can be achieved by each agent applying a digital signature to each p-assertion. The user can validate a p-assertion's signature when it retrieves it from a provenance store (the store may also do its own checks).

This guards against a particular type of deception that applies to both transparent and exploitable systems: an agent may assert something false but attempt to make it appear that the assertion comes from another, trusted, agent. Without accountability, agents are free to give a completely false view of a process without detection.

5 Applying AgentPrIME

Applying the AgentPrIME methodology fragment in the context of an agent-oriented methodology requires that the developer knows both at what point to apply it and what steps to take in doing so. With regard to the first point, we have already said that AgentPrIME is ideally applied at a particular point in the extended methodology, when the design is sufficiently well developed to adapt but not so far developed that effort is wasted. In practice, given the adaptations described above, this means that (types of) agents have been defined well enough to know the (types of) interactions they will take part in and the causal chains their actions lead to. Depending on the methodology, some adaptations may be best applied even later, when an agent's internal structure is defined.

Once a reasonable point in the methodology has been decided on, the developer, at that point in a system's design, should consider each agent in turn and determine how to 'wrap' the agent to record p-assertions about its activity into a provenance store, preferably a third party one. The form that such wrapping takes depends on technology: the aim is for the agent's logic to trigger the recording of p-assertions and anything which achieves this aim is considered to be an instantiation of a wrapper adaptation. The consideration of ensuring accuracy can then begin. First, where possible, an agent should be adapted to sign its p-assertions. Second, each interaction between agents should be considered and, where a third party would not be able to corroborate the contents of the interaction, a third party should be added and interactions redirected through it. The choice of third party is based on the developers' best guess as to what will be trusted by those others the system is likely to interact with.

In terms of tool support, if an agent's internal operations are made explicit, e.g. as an architecture plus plans, then it may be possible to automate the modelling of causation in that agent.

6 Conclusions

AgentPrIME is an extension applicable to existing agent-oriented methodologies that gives users confidence in the results produced by designed systems. Developers applying AgentPrIME to a design will determine how that design needs to be adapted to, firstly, record adequate documentation that exposes the qualities of the process that produced some output of the system, and then to ensure that the documentation itself is reliable through corroboration, independent storage and accountability of agents.

The approach aims to be as *methodology neutral* as possible, being applicable regardless of the agent-oriented concepts that have been used in designing a system. It does this by defining *causal relationships*, that which must be documented, in a system-independent way, and by relying only on the agents and their interactions, which are present in any multi-agent system.

Four design adaptations are defined in this paper:

Wrapper Adaptation Adapting agents (or agents of a given type) to record documentation on what they have done and why.

Corroboration Adaptation Adapting agent interactions that may be seen as collusion so that an intermediary can provide collaborating evidence of the communications.

Third-Party Storage Adaptation Providing storage of documentation that is trusted by both recording agents and users.

Accountability Adaptation Adapting agents to sign data before recording it for users to query.

In future work, we will investigate the further uses of the process documentation recorded by a multi-agent system. For instance, it may be possible to determine whether agents have fulfilled their responsibilities and do not prevent other agents exercising their rights, by examining the documentation recorded. We will also explore the integration of AgentPrIME with trust models [14], which can provide sophisticated assessments of agents' reliability.

References

1. Christopher Cheong and Michael Winikoff. Hermes: Designing goal-oriented agent interactions. In *Agent-Oriented Software Engineering VI: 6th International Workshop (AOSE 2005)*, pages 16–27, Utrecht, Netherlands, 2005.
2. Scott A. DeLoach, Mark F. Wood, and Clint H. Sparkman. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231–258, 2001.
3. Alessandro Garcia, Uira Kulesza, Claudio Sant'Anna, Christina Chavez, and Carlos J. P. de Lucena. Aspects in agent-oriented software engineering: Lessons learned. In *Agent-Oriented Software Engineering VI: 6th International Workshop (AOSE 2005)*, pages 231–247, Utrecht, Netherlands, 2005.

4. Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. An architecture for provenance systems. Technical report, Electronics and Computer Science, University of Southampton, October 2006. Available at <http://eprints.ecs.soton.ac.uk/12023/>.
5. Paul Groth, Simon Miles, Weijian Fang, Sylvia C. Wong, Klaus-Peter Zauner, and Luc Moreau. Recording and using provenance in a protein compressibility experiment. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC'05)*, pages 201–208. IEEE Computer Society, July 2005.
6. S. Guttenplan. *Introduction to Philosophy of Mind*, chapter An Essay on Mind. Oxford University Press, 1994.
7. Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley, 2004.
8. Ivan J. Jureta, Stephane Faulkner, and Pierre-Yves Schobbens. Allocating goals to agent roles during mas requirements engineering. In *Proceedings of the 7th International Workshops on Agent-Oriented Software Engineering' (AOSE 2006)*, 2006. Forthcoming.
9. Kuldeep Kumar and Richard J. Welke. Methodology engineering: a proposal for situation-specific methodology construction. In *Challenges and strategies for research in systems development*, pages 257–269, New York, NY, USA, 1992. John Wiley & Sons, Inc.
10. Simon Miles, Steve Munroe, Michael Luck, and Luc Moreau. Modelling the provenance of data in autonomous systems. In *Proceedings of Autonomous Agents and Multi-Agent Systems 2007*, page 8 pages, Honolulu, Hawai'i, May 2007.
11. Steve Munroe, Simon Miles, Luc Moreau, and Javier Valquez-Salceda. Prime: A software engineering methodology for developing provenance-aware applications. In *Proceedings of the Software Engineering and Middleware Workshop (SEM 2006)*. ACM Digital, 2006. To appear.
12. Andrea Omicini. SODA: Societies and infrastructures in the analysis and design of agent-based systems. In *Proceedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*, pages 185–193, 2000.
13. Lin Padgham and Michael Winkoff. Prometheus: a methodology for developing intelligent agents. In *Agent-Oriented Software Engineering III: Third International Workshop (AOSE 2002)*, pages 174–185, Bologna, Italy, July 2002.
14. W. T. L. Teacy, J. Patel, N. R. Jennings, and M. Luck. Travos: Trust and reputation in the context of inaccurate information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 12(2):183–198, 2006.
15. Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000.