Programming Multiagent Systems without Programming Agents

Munindar P. Singh[1] and Amit K. Chopra[2]

[1] North Carolina State University singh@ncsu.edu
[2] Università degli Studi di Trento akchopra.mail@gmail.com

Abstract. We consider the programming of multiagent systems from an architectural perspective. Our perspective emphasizes the autonomy and heterogeneity of agents, the components of multiagent systems, and focuses on how to specify their interconnections in terms of high-level protocols. In this manner, we show how to treat the programming of a multiagent system as an architectural endeavor, leaving aside the programming of individual agents who might feature in a multiagent system as a secondary concern.

1 Introduction

This paper presents a new way of thinking about the programming of multiagent systems. Most existing approaches either seek to apply traditional software engineering or to apply traditional artificial intelligence metaphors and abstractions. In contrast, this paper takes a uniquely multiagent systems perspective. It focuses on how to describe the interactions among agents in a manner that facilitates their loose coupling, and thus naturally respects their autonomy and heterogeneity.

Like traditional software engineering approaches, this paper gives primacy to interfaces and contracts. But unlike traditional approaches, it formulates these at a high level. Like traditional artificial intelligence approaches, it considers high-level abstractions that need not make sense in all applications and specifically are not pursued in traditional software engineering. Unlike traditional artificial intelligence, it gives prominence to social and organizational abstractions as opposed to cognitive ones, and offers a way to judge the compliance of agents.

Before we talk about what constitutes a multiagent architecture, it is helpful to consider how architecture fits into software engineering and how we understand it here. An *architecture* is motivated by requirements of the stakeholders of the systems that instantiate it as well as by the environment in which it is instantiated [1]. Traditional engineering draws an important distinction between functional and nonfunctional requirements. The former deal with functionality that is somehow relevant to the problem domain—for example, a sorting service would differ from a matrix inversion service on functional grounds. The latter deal with aspects of how that functionality is delivered—for example, with what latency, throughput, and availability. The associated idea in traditional engineering is that all approaches would meet the functional requirements but architectures would largely vary based on the nonfunctional requirements that they support [2].

However, the above distinction—although a useful one—is far from perfect. It is not always clear, as is well known, how to distinguish functional from nonfunctional requirements. For example, if we are relying upon a numerical computation service to determine how much to decelerate an automobile so it avoids a collision, the apparently nonfunctional requirement of latency is very much functionally critical.

More importantly, when we think of a multiagent system in the broad sense, it is not at all clear whose requirements we are dealing with. A traditional software system would usually have multiple stakeholders: some users, some user advocates, some administrators, some developers (including, for us, designers, implementers, and maintainers). The developers are a bit of an outlier in this list in that they are not users of the system but they do impose requirements such as the maintainability of a system, which arguably an end user has no direct interest in—at least within a usage episode. However, when end users have an interest in having a system keep up with evolving requirements, maintainability becomes key to them as well. Regardless, the various stakeholders negotiate (perhaps in absentia via the developers) to determine and prioritize system requirements. The ultimate product—the system—is a tightly integrated whole that ought to meet its (suitably negotiated) requirements.

In sharp contrast with a traditional software system, it is generally not appropriate to think of a multiagent system as being designed in its totality to serve one integrated set of requirements. This is because in a typical multiagent system, the stakeholders are autonomous entities and do not necessarily serve the interests of a common enterprise. Many of the stakeholders are projected into the designed system as autonomous entities, that is, as agents. These agents are generally heterogeneous, meaning that they not only exhibit diverse designs and implementations but also instantiate apparently idiosyncratic decision policies.

It is worth emphasizing this point further. We are concerned with the programming of multiagent systems that not only involve multiple autonomous stakeholders, but also keep those stakeholders isolated from one another in the sense that the stakeholders may potentially hold divergent stakes in the different components. Service-oriented applications—banking, auctions, flight reservation, e-business in general, e-health, foreign exchange transactions, and so on—are prime examples of such systems; so are normative systems and virtual organizations. There are indeed multiagent systems, especially in cooperative applications involving distributed sensing, teamwork, and so on, that resemble a tightly integrated whole in the sense described above. For such applications, the system is decomposed into multiple agents because of some feature of the environment, such as the distributed nature of the information to be obtained or actions to be performed [3], or simply to facilitate a separation of concerns among different active modules [4]. Such applications do not emphasize the autonomous nature of agents and thus are not of primary concern here. In the following, the term *multiagent systems* refers exclusively to systems with multiple stakeholders, at least some of whom are isolated.

For the above reasons, traditional software architectures and their concomitant design methodologies are not readily applicable to the design and implementation of multiagent systems. Our present interest is to consider the aspects of multiagent systems that are *unique* to multiagent systems in the realm of software engineering [5]. For this reason, we claim that any approach that focuses on traditional programming artifacts works at too low a level to be of great value. In the same vein, any approach that focuses on building an integrated solution is generally inapplicable for multiagent systems. In contrast to integration, we seek approaches that emphasize the interoperation of autonomous and heterogeneous components.

Consequently, we advocate an approach for programming multiagent systems that does not look within individual agents at all. Instead, this approach talks about the interactions among agents. The interactions could themselves be captured through fiat in the worst case, through design-time negotiation among some of the stakeholders, or through runtime negotiation among the participating agents (based ultimately on some design-time negotiation at least to nail down the language employed within the negotiation). Our primary focus here is on the middle category above although the concepts we promote can work in the other categories as well.

2 Architecture in General

Let us begin with a brief study of an *architecture* in conceptual terms. Understood abstractly, an architecture is a description of how a system is organized. This consists primarily of the ingredients of the system, that is, its *components* and the *interconnections* it supports among the components. An *architectural style* is an abstraction over an architecture. A style identifies the following:

– *(Architectural) Constraints* on components and interconnections.
– *Patterns* on components and interconnections.

An architectural style yields a description language (possibly, also a graphical notation) in which we can present the architectures of a family of related systems and also the architecture of a particular system.

An *open architecture* is one whose components can change dynamically. Therefore, the openness of an architecture arises from its specifying the interconnections cleanly. In other words, the *physical* components of the architecture all but *disappear*; in their stead, the *logical* traces of the components remain. We define *protocols* as the kinds of interconnections that arise in open information environments.

2.1 Criteria for Judging Interconnections

The purpose of the interconnections is to support the interoperation of the components that they connect. How may we judge different kinds of interconnections? Our assessment should depend upon the level of interoperation that the interconnections support.

In particular, given our motivation for multiagent systems in Section 1, we identify the following criteria.

– *Loose coupling:* support heterogeneity and enables independent updates to the components.
– *Flexibility:* support autonomy, enabling participants to extract all the value they can extract by exploiting opportunities and handling exceptions.

– *Encapsulation:* promote modularity, thereby enabling independent concerns to be modeled independently, thus facilitating requirements tracing, verification, and maintainability.
– *Compositionality:* promote reuse of components across different environments and contexts of usage, thereby improving developer productivity.

We take the view that two or more components *interoperate* when each meets the expectations that each of the others places on it. An important idea—due to David Parnas from the early days of software architecture—is that interoperation is about each component satisfying the assumptions of the others [6]. Parnas specifically points out that interoperation is neither about control flow nor about data flow.

Unfortunately—and oddly enough—most if not all, subsequent software engineering research considers *only* control or data flow. As we explained in the foregoing, such approaches emphasize low-level abstractions that are ill-suited for multiagent systems. However, considering expectations abstractly and properly opens up additional challenges. Specifically,

– How may we characterize the expectations of components regarding each other except via data and control flow?
– How may we verify or ensure that the expectations of a component are being met by the others?

2.2 Protocols, Generally

The main idea is that a protocol encapsulates the interactions allowed among the components. In this sense, a protocol serves two purposes. On the one hand, a protocol *connects* components via a conceptual interface. On the other hand, a protocol *separates* components by providing clean partitions among the components viewed as logical entities. As a result, wherever we can identify protocols, we can (1) make interactions explicit and (2) identify markets for components. That is, protocols yield standards and their implementations yield products.

Let us consider protocols in the most general sense, including domains other than software, such as computer networking or even power systems. In networking, conventional protocols such as IEEE 802.11g and the Internet Protocol meet the above criteria. They determine what each component may expect of the others. They help identify markets such as of wireless access points and routers. In power systems, example protocols specify the voltage and frequency an electrical component can expect from a power source and the ranges of acceptable impedances it must operate within.

3 Proposed Approach

What are some key requirements for an architecture geared toward multiagent systems? Clearly, the components are *agents*, modeled to be *autonomous* (independent in their decision making) and *heterogeneous* (independently designed and constructed). Further, the environment of a multiagent system provides support for

– Communication: inherently *asynchronous*.
– Perceptions.
– Actions.

For Information Technology environments, we can treat all of the above as communications. The key general requirement for a multiagent system is that its stakeholders require the agents to *interoperate*. The specifics of interoperation would vary with the domain. In our approach, these would be captured via the messages and their meanings that characterize the interactions among the members of the desired multiagent system.

3.1 Specifying Multiagent System Protocols

In light of the above criteria, we can approach the problem of specifying multiagent system protocols in the following main ways. Following traditional methodologies, we can take a *procedural* stance, which would specify the *how* of the desired interaction. Examples of these approaches that are well-known even in current practice include finite state machines and Petri nets. In general, the procedural approaches over-specify the desired interactions, thus limiting flexibility and coupling the components more tightly than is necessary.

Alternatively, we can take a *declarative* stance, which would specify the *what* of the desired interaction, meaning what it seeks to accomplish. Examples of declarative approaches are those based on the various forms of logic: predicate, modal, temporal, and such. A logic-based approach is not necessarily higher level than the procedural approaches. What matters primarily or even solely is what the conceptual model is that the formalization seeks to capture. We advocate declarative approaches based on high-level conceptual abstractions that promote loose coupling and flexibility.

Our proposed approach can be summarized as *agent communication done right*. Communication in general and agent communication in particular can be understood as involving at least the following main aspects of language.

– *Syntax:* documents to be exchanged as messages. We can imagine these documents as being rendered in a standardized notation such as a vocabulary based on XML.
– *Semantics:* formal meaning of each message. We propose that at least for business applications, this desired meaning may be best expressed using abstractions based on the notion of *commitments* [7]. For other situations involving multiagent systems, the meaning could potentially be expressed via other suitable constructs in a like manner. However, even in nonbusiness settings, the commitments among agents can be valuable. In particular, the type of commitment known as dialectical may be suitable for applications involving the exchange of information or arguments, such as negotiation [8].

Our approach places considerable weight on the idea of minimizing operational constraints. Our motivation for this is to enhance the features listed in Section 2.1, specifically loose coupling, which promotes autonomy and heterogeneity. From our experience in formalizing various domains, it is worth remarking—and it would prove surprising to many—that few operational constraints are truly needed to capture the essential requirements of an application.

3.2 Ensuring Interoperation via Traditional Representations

Traditional software engineering approaches apply only at the level of control and data flow among the components. Usually the concomitant flows are specified procedurally, although they can be captured declaratively. Regardless, as we argued in Section 2.1, control and data flows prove to be a poor notion of interoperation for multiagent systems. However, it is important to recognize that traditional approaches *can support* interoperation, albeit at a low level. Further, they carry a concomitant notion of compliance as well.

In contrast, the traditional agent-oriented approaches—based as they are on traditional artificial intelligence concepts—place onerous demands on the agents. Because these approaches emphasize the cognitive concepts such as beliefs, goals, desires, or intentions, they presuppose that the agents be able to interoperate at the cognitive level. In other words, the traditional agent-oriented approaches require that the cognitive state of an agent be

- Externally *determinable*, which is impossible without violating the heterogeneity of the agents.
- In constant *mutual agreement*, which is impossible without violating autonomy and asynchrony.

Consequently, we claim that these approaches offer no viable notion of interoperation (or of compliance [9]). In this manner, they reflect a step backwards from the traditional software engineering approaches.

For the above reasons, the BDI approaches are *not* suitable for architecture. They (1) violate heterogeneity by presuming knowledge of agent internals; (2) prevent alignment in settings involving asynchrony; (3) tightly couple the agents with each other; and (4) lead to strong assumptions such as sincerity in communication that prove invalid in open settings.

3.3 Interoperation via Commitments

Commitments yield a notion of compliance expressly suited for multiagent systems. Agent compliance amounts to the agent not violating any of its commitments towards others. A protocol specified in terms of commitments does not dictate specific operationalizations in terms of when an agent should send or expect to receive particular messages; as long as the agent discharges its commitments, it can act as it pleases [10].

We introduce some notation and elementary reasoning rules for commitments.

- The expression $\mathsf{C}(\mathsf{debtor}, \mathsf{creditor}, \mathsf{antecedent}, \mathsf{consequent})$ represents a commitment; it means that debtor is committed to the creditor for the consequent if the antecedent is brought about. For example, $\mathsf{C}(\mathsf{EBook}, \mathsf{Alice}, \$12, \mathsf{BNW})$ means that EBook is committed to Alice for the book *BNW* (for *Brave New World*) in return for a payment of \$12.
- $\mathsf{C}(\mathsf{debtor}, \mathsf{creditor}, \top, \mathsf{consequent})$ represent an unconditional commitment. For example, $\mathsf{C}(\mathsf{EBook}, \mathsf{Alice}, \top, \mathsf{BNW})$ means that EBook is committed to Alice for the book *BNW*.

– DETACH: $\mathsf{C}(x, y, r, u) \wedge r \rightarrow \mathsf{C}(x, y, \top, u)$: if the antecedent holds, then the debtor become unconditionally committed to the consequent. For example (reading $\Rightarrow$ as logical consequence), $\mathsf{C}(\mathsf{EBook}, \mathsf{Alice}, \$12, \mathsf{BNW}) \wedge \$12 \Rightarrow \mathsf{C}(\mathsf{EBook}, \mathsf{Alice}, \top, \mathsf{BNW})$.
– DISCHARGE: $u \rightarrow \neg\mathsf{C}(x, y, r, u)$: if the consequent holds, the commitment is *discharged*—it does not hold any longer—no matter if it is conditional or not. For example, both of the following hold. $\mathsf{BNW} \Rightarrow \neg\mathsf{C}(\mathsf{EBook}, \mathsf{Alice}, \$12, \mathsf{BNW})$. And, $\mathsf{BNW} \Rightarrow \neg\mathsf{C}(\mathsf{EBook}, \mathsf{Alice}, \top, \mathsf{BNW})$

The flexibility of a (complying) agent is limited by the need to interoperate with others. To fully exploit the flexibility afforded by commitments, we must necessarily formalize interoperability in terms of commitments. For if we continued to rely upon the notions of interoperability as formalized for components—in terms of a component being able to simply send and receive messages as assumed by others—we would be introducing operational constraints on the communication among agents, thus limiting their flexibility.

To motivate our definition of interoperation in terms of commitments, we observe that there are two primary sources of asymmetry in a multiagent system. On the one hand, communications are inherently directed with the direction of causality usually being treated as flowing from the sender to a receiver (but see Baldoni *et al.* [11] for a more general, alternative view). On the other hand, commitments are directed with the direction of expectation being from the creditor of a commitment to its debtor.

Accordingly, we propose a notion of interoperation that we term *(commitment) alignment* [12]. Alignment, as we define it, is fundamentally asymmetric. The intuition it expresses is that whenever a creditor computes (that is, infers) a commitment, the presumed debtor also computes the same commitment. The formalization of this definition involves some subtlety, especially on the notion of what we mean by *whenever*. Specifically, we capture the intuition that at the moments or system snapshots at which we judge the alignment or otherwise of any two agents, we make sure that the agents have received the same relevant information. Thus all messages sent must have been received, and each agent ought to have shared any information it has received that is materially relevant to the commitments in which it participates. In particular, a creditor should propagate information about partial or total detachments, which strengthen a commitment. And, a debtor should propagate information about partial or total discharges, which weaken or dissolve a commitment.

In this manner, our approach can achieve alignment even in the face of asynchrony—meaning unbounded message delays but messaging that is order-preserving for each pair of sender and receiver. The approach works as follows. When a debtor autonomously creates a commitment, it sends a corresponding message, which eventually lands at the creditor. Here the debtor is committed before the creditor learns of the debtor being committed, so alignment is preserved. When a creditor detaches a commitment, thereby strengthening it, a message corresponding to the detach eventually arrives at the debtor. Here the debtor is committed when it receives the detach message.

The foregoing motivates a treatment of *quiescence* wherein we only consider well-formed points in executions where each message has landed. When a debtor or creditor learns that a commitment is discharged or detached, respectively, it must immediately notify the other (*integrity*, which ensures no quiescence until the information has propagated).

In broad terms, our ongoing research program calls for the development of what we term *CSOA*, a commitment-based service-oriented architecture [13]. CSOA is focused on the notion of a *service engagement*. When thought of in business terms, a service engagement involves multiple business partners carrying out extensive, subtle interactions in order to deliver value to each other. Business services are to be contrasted with technical services such as on Web or the Grid, which emphasize lower level questions of connectivity and messaging without regard to the business content of the interactions.

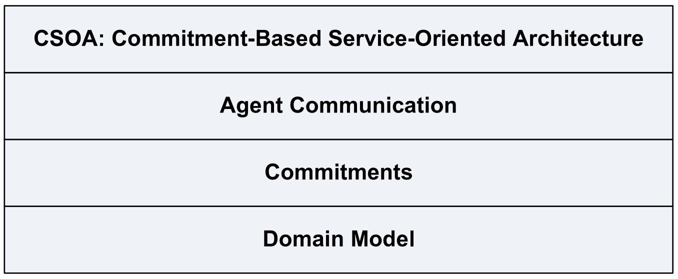

Fig. 1. Proposed architecture schematic, conceptually

Figure 1 shows how we imagine the layers of our architecture in conceptual terms. For a given application, the *domain model* describes the roles involved and the vocabulary, including the business documents that agents adopting the role would exchange. The *commitments* layer understands the documents in terms of their business contents. The *agent communication* layer deals with the primitive commitment operations such as Create, Delegate, and so on, and other communication primitives such as Request, Declare, and so on; the primitives in this layer would be more or less standard. Finally, the *CSOA* layer deals with composite service engagement patterns built from the primitive commitment operations. For example, a book-selling application would involve at least the roles $Buyer$ and $Seller$. An *Offer* for some book from the $Seller$ could be mapped to a commitment from the $Seller$ to the $Buyer$, similar to the one above from EBook to Alice. A CSOA pattern for the book-selling application could encode refunds from the $Seller$.

4 Programming Multiagent Systems

Based on the foregoing, we can now introduce a conceptually straightforward way in which to program a multiagent system.

4.1 Step 1: Specify the Communications

We define a protocol and specify it as follows.

– Specify roles to describe abstracted versions of the agents who will participate in the multiagent system at runtime.
– Specify messages as the surface communications among various pairs of roles.
– Specify the meanings of each message declaratively in terms of commitments and other relevant propositions.
– Specify any additional constraints on the messages such as the conventions of the relative orders among the messages, and how information carried in one message flows to another message in the protocol.

The above commitment-based specification approach is reasonable for the following reasons. Based on the notion of commitments, we can unambiguously determine if a particular enactment satisfies the specified protocol or not. We can also determine if any of the agents is noncompliant. Further, if needed, we can refine and compose protocols to produce protocols that better address our stakeholder requirements.

4.2 Step 2: Instantiate the System

The next step in this methodology is to instantiate and configure a multiagent system so as to be able to enact its computations. To instantiate and enact a multiagent system, identify agents to play roles in the protocol that characterizes the multiagent system. We refer to a unique protocol because when there are multiple protocols, they can be considered for this discussion as having been composed into a single definitive protocol. In practice, we do not require the creation of a monolithic composed protocol. The instantiation of a multiagent system could proceed one of three ways in terms of the agents who are involved. These agents could be any combination of (1) preexisting agents proceeding on their own initiative; (2) newly instantiated agents based on preexisting code-bases; and (3) custom-designed agents to suit the needs of the stakeholders who contribute them to the multiagent system. Different stakeholders could follow any of the above approaches in constructing the agents they field in the given system. In any case, each agent would apply the decision-making policies of the stakeholder whom it represents computationally within the multiagent system.

4.3 Enactment and Enforcement

The agents collectively enact this programming model by individually applying their policies to determine what messages to send each other. As explained above, the meaning of each message specifies how it corresponds to operations on commitments.

The above approach can be readily realized in runtime tools, which can be thought of as commitment middleware [14]. The middleware we envisage would offer primitives encapsulated as programming abstractions by which each agent can

– Communicate with other agents.
– Maintain the commitments in which it features as debtor or creditor.
– Propagate the information necessary to maintain alignment among the agents.
– Verify the compliance of debtors with commitments where it is the creditor.

Such a middleware would enable writing programs directly in terms of commitments. Instead of an agent effecting communication with others through such low-level primitives as *send* and *receive*, the agent would perform commitment operations.

Our commitment-based architecture does not require that there be a central authority to enforce commitments. In general, in settings with autonomous agents, no agent can be sufficiently powerful to force another agent to act in a certain manner. Enforcement in such settings realistically amounts to arbitration and applying penalties where appropriate.

A commitment is made in a certain *context*, which defines the rules of encounter for the agents who feature in it [13]. We model the context as an agent in its own right. Viewed in this light, the context can perform several important functions, not necessarily all of them in the same setting. The context can be a monitor for tracking commitments, which assumes it observes all communications. Alternatively, the context may serve as arbiter for any disputes between the contracted parties.

In some settings, the context may also act as a sanctioner who penalizes agents who violate their commitments. The context may cause a new *penalty* commitment (with the same debtor as the violated commitment) to come into force. Ultimately, there is little the context can do, except possibly to eject a malfeasant agent from the interaction. The context may observe the violation itself or may learn of it from the creditor, who would have escalated the commitment to the context. For example, a buyer on eBay (the marketplace) may escalate a dispute with a seller to eBay (the corporate entity, serving as the context). The protocol for escalating and dispute resolutions may be considered as part of a larger Sphere of Commitment [15]. However, often, penalties are left unspecified. For example, a buyer's agent may simply notify the buyer that the seller has violated some commitment, at which point the buyer may take up the matter with eBay. A more common approach is to use reputation as a form of social censure for malfeasant agents. In well-structured settings such as eBay, which might support a notion of reputation, there is also the option of ejecting a malfeasant agent.

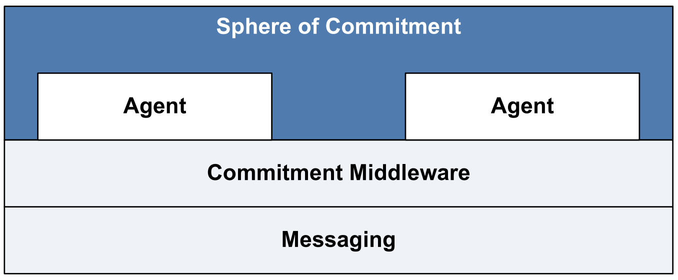

Fig. 2. Proposed architecture schematic, operationally

Figure 2 illustrates a way to operationalize our architecture in schematic terms. A commitment middleware resides above messaging and makes sure that agents maintain their alignment by exchanging relevant information. The agents function within a suitably powerful sphere of commitment which, as explained above, potentially ensures they comply with their commitments.

4.4 Summary of Benefits

We highlight the benefits of our approach according to the criteria presented in Section 2.1. Formalizing interoperability in terms of commitment alignment promotes a *looser* coupling among agents than is possible with traditional approaches. In particular, many of the message ordering constraints that are typically taken for granted in real-life applications, for example, that the *accept* or the *reject* of an *offer* must follow the *offer*, are no longer necessary. In effect, when we loosely couple agents, they can update their commitments independently of each other.

Commitments support a high-level notion of compliance, and thus support flexible enactment. In earlier work on commitments, the flexibility afforded by commitments could not be fully exploited as concerns of concurrency obscured the picture somewhat. With interoperation formalized in terms of commitments, agents can fully exploit this flexibility.

Encapsulation and compositionality have to do with the efficient software engineering of protocols [16]. In essence, each protocol is a discrete artifact, independent from requirements and from other protocols. A protocol may thus be made available in a repository, and depending on a particular application's requirements, composed with other protocols and instantiated.

5 Discussion: Conclusions and Future Work

First, we observe that existing multiagent systems engineering approaches, in attempting to develop practical systems, adopt traditional software engineering ideas wholesale. In this manner, they tend to neglect the key features that characterize multiagent systems, specifically, the autonomy and the heterogeneity of their participants.

Second, when existing approaches recognize the high-level nature of the descriptions of agents and their interactions, they seek to differentiate themselves from traditional software engineering by introducing concepts from traditional artificial intelligence, specifically, concepts such as beliefs, goals (or desires), and intentions. In this manner, they continue to poorly accommodate the asynchrony, autonomy, and heterogeneity that characterize real-life multiagent systems.

We advocate an approach in which high-level concepts yield interconnections that support multiagent system applications. These concepts are centered on commitments and help model the interactive nature of multiagent systems directly. A key challenge is that we realize such concepts correctly in order to achieve interoperation.

The key to building large-scale multiagent systems lies in adequately formalizing agent communication, not the internal decision making of agents. Whether an agent is capable of reasoning about beliefs or whether the agent is specified as an automaton is neither relevant nor discernible to another agent. A desirable development would be if both agent communication and reasoning could be specified in terms of high-level abstractions, and the runtime infrastructure would directly support the abstractions. This would obviate the need to translate between different levels of abstraction, as advocated in model-driven approaches, and would truly usher in the age of agent-oriented software engineering. For this, we would have to formally relate agent reasoning with agent communications. This challenge is beginning to be addressed in the recent literature [17, 18].

Considerations of multiagent systems require fresh approaches in requirements modeling. Instead of classifying requirements simply as functional or nonfunctional, one also needs to consider whether the requirement is *contractual*—implying a commitment between two of the stakeholders—or noncontractual. A requirement could be functional and contractual (for example, EBook's offer entails such a commitment), or nonfunctional and contractual (for example, the requirement that the book be delivered using priority service), and so on. Indeed, as pertains to multiagent systems, the contractual dimension seems to be more significant than the functional one.

In business engagements, the context plays an important role—it instills some measure of confidence in compliance by the interacting parties. In commitments of a personal nature, the context may be implicit. Further, there may not be any explicitly specified penalty commitments. For example, if Rob commits to picking up Alice from the airport, but does not show up on time, Alice may cancel her dinner engagement with him or she may simply note Rob to be unreliable. The point to be taken here is that commitments are valuable because they enable reasoning about compliance; their value does not derive from their being enforced or not. Neither penalty nor arbitration are semantically integral to commitments.

5.1 A Remark on Notation

Architectural approaches inherently lead to ways to describe systems. Thus they naturally lead and should lead to notations. Notation, although important, remains secondary to the concepts. When we describe an architecture, what matter most are the concepts using which we do so. It is more important to develop a suitable metamodel than to specify a detailed notation that lacks an appropriate metamodel.

We notice a tendency in agent-oriented software engineering where, in attempting to develop practical systems, researchers adopt traditional notations wholesale as well. There is indeed value in adopting traditional notations, but *only* where such notations apply. The field of multiagent systems exists—and research into the subfield of programming multiagent systems is a worthwhile endeavor—only because traditional approaches are known to be inadequate for a variety of practical information systems, especially large-scale open, distributed systems. In other words, existing notations are not complete for these purposes. Therefore, a worthwhile contribution of multiagent system research is to invent suitable notations backed up by expressive metamodels.

5.2 Directions

CSOA is an architecture style that treats business (not technical) services as agents, and includes patterns for service engagements. Along the lines of CSOA, we have recently begun to develop a business modeling language [17]. This language is based on a metamodel that provides first-class status to business partners and their respective commitments, expressing their contracts. It also includes support for some CSOA patterns such as for delegating commitments that are core to the precise, yet high-level specification of a service engagement.

Upcoming research includes a study of ways in which to express a multiagent system in terms of the business relationships among agents as conglomerates of commitments, and formal methods to verify the computations realized with respect to business models.

Acknowledgments

Munindar Singh's research was partially supported by the Intel Research Council. Amit Chopra's research was partially supported by the Integrated FP6-EU project SERENITY contract 27587.

References

1. Zachman, J.A.: A framework for information systems architecture. IBM Systems Journal **26**(3) (1987) 276–292
2. Filman, R.E., Barrett, S., Lee, D.D., Linden, T.: Inserting ilities by controlling communications. Communications of the ACM **45**(1) (2002) 116–122
3. Durfee, E.H.: Practically coordinating. AI Magazine **20**(1) (Spring 1999) 99–116
4. Singh, M.P., Huhns, M.N.: Automating workflows for service provisioning: integrating AI and database technologies. IEEE Expert **9**(5) (October 1994) 19–23
5. Huhns, M.N., Singh, M.P., Burstein, M.H., Decker, K.S., Durfee, E.H., Finin, T.W., Gasser, L., Goradia, H.J., Jennings, N.R., Lakkaraju, K., Nakashima, H., Parunak, H.V.D., Rosenschein, J.S., Ruvinsky, A., Sukthankar, G., Swarup, S., Sycara, K.P., Tambe, M., Wagner, T., Gutierrez, R.L.Z.: Research directions for service-oriented multiagent systems. IEEE Internet Computing **9**(6) (November-December 2005) 65–70
6. Parnas, D.L.: Information distribution aspects of design methodology. In: Proceedings of the International Federation for Information Processing Congress. Volume TA-3., Amsterdam, North Holland (1971) 26–30
7. Singh, M.P.: An ontology for commitments in multiagent systems: Toward a unification of normative concepts. Artificial Intelligence and Law **7**(1) (March 1999) 97–113
8. Singh, M.P.: Semantical considerations on dialectical and practical commitments. In: Proceedings of the 23rd Conference on Artificial Intelligence (AAAI), Chicago, AAAI Press (July 2008) 176–181
9. Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer **31**(12) (December 1998) 40–47
10. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: Applying event calculus planning using commitments. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems, Bologna, ACM Press (July 2002) 527–534
11. Baldoni, M., Baroglio, C., Chopra, A.K., Desai, N., Patti, V., Singh, M.P.: Choice, interoperability, and conformance in interaction protocols and service choreographies. In: Proceedings of the 8th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), Budapest, IFAAMAS (May 2009) 843–850
12. Chopra, A.K., Singh, M.P.: Multiagent commitment alignment. In: Proceedings of the 8th International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS), Budapest, IFAAMAS (May 2009) 937–944

13. Singh, M.P., Chopra, A.K., Desai, N.: Commitment-based service-oriented architecture. IEEE Computer **42**(11) (November 2009)
14. Chopra, A.K., Singh, M.P.: An architecture for multiagent systems: An approach based on commitments. In: Proceedings of the 7th International Workshop on Programming Multiagent Systems (ProMAS). LNCS, Springer. This volume.
15. Singh, M.P.: Multiagent systems as spheres of commitment. In: Proceedings of the International Conference on Multiagent Systems (ICMAS) Workshop on Norms, Obligations, and Conventions. (December 1996)
16. Desai, N., Chopra, A.K., Singh, M.P.: Amoeba: A methodology for modeling and evolution of cross-organizational business processes. ACM Transactions on Software Engineering and Methodology (TOSEM) **19**(2) (October 2009) 6:1–6:45
17. Telang, P.R., Singh, M.P.: Business modeling via commitments. In: Proceedings of the 7th AAMAS Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE). LNCS **5907**, Springer
18. Robinson, W.N., Purao, S.: Specifying and monitoring interactions and commitments in open business processes. IEEE Software **26**(2) (March 2009) 72–79