Federico Bergenti
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Parco Area delle Scienze 181/A, 43100 Parma, Italy
bergenti@ce.unipr.it

Michael N. Huhns
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC 29208, USA
huhns@sc.edu

On the Use of Agents as Components of Software Systems

Software agents are increasingly being used as building blocks of complex software systems. In this paper we discuss the benefits and the drawbacks that a developer faces when choosing agents for the realization of a new system, instead of a more mature technology such as software components. In particular, we first compare agents to components and then highlight the differences and similarities engendered by the metaphors and abstractions that each provides. Then, we concentrate our comparison on reusability because of general agreement that reusability is one of the most important features to consider when adopting a development technology. We exploit agent-oriented concepts to define formally an asymptotic level of reusability, and we show how agents and components approximate it. The result of such a comparison is that agents are intrinsically more reusable than components.

1. Introduction

The creation and development of Agent-Oriented Software Engineering (AOSE) [13] in the last decade has promoted agents as a viable new way to develop complex software systems. AOSE gives to the developer all the flexibility and the expressive power of agents and it helps with the management of the software lifecycle in an attempt to improve the quality of the resultant software products.

During its short history, research on AOSE has undergone an important change of focus: initially it was meant only to provide methodologies and tools to build agent-based systems; today it is more concentrated on understanding the features that an agent-based approach can bring to the development of conventional software. This change of focus is not trivial and it corresponds to a radically different approach in adopting agents during the evolution of the software lifecycle. The first approach is based on choosing agents as the very basic abstraction of the development, before actually starting the software lifecycle. Such a decision is taken for reasons that fall outside of the software lifecycle and is generally based on the nature of the system or on the complexity of the problem at hand.

Choosing agents as the very basic abstraction for development allows adopting the agent-oriented mindset for the entire lifecycle, from analysis of early requirements to the retirement of the system, as the Tropos methodology [6] suggests. The major drawback of this is that often there is no reasonable motivation for choosing agents to develop conventional systems, e.g., word processors and financial planning systems. This is the reason why a more modern approach to AOSE tends to move the decision on adopting agents after (or during) the phases for requirement analysis and requirement specification. The developer is not forced to envisage his/her system in terms of agents, rather he/she can concentrate on the requirements that will drive the subsequent design. The major disadvantage of this approach is that the developer may not exploit interesting features of agents, e.g., emergent behavior and generalization, because he/she has concentrated too much on the concrete requirements that come directly from the client.

Besides this drawback, the more recent approach has highlighted the problem of understanding when and how the developer should prefer agent technology instead of any more traditional, and possibly more mature, technology, such as object-oriented technology. In this paper we address a particular aspect of this problem and we present the motivations for choosing agents instead of a technology that resembles agents from many points of view: software components [22]. In the following section we compare agents and components by taking into account five aspects that they share. Our results have maximal generality, because our analysis is not bound to a particular technology. In section 3 we extend the depth of our comparison by concentrating on reusability, which is one of the most important aspects of a development technology. We begin this by formally defining an asymptotic level of reusability, and then show how agents and components approximate it. The result of such a comparison is that agents are intrinsically more reusable than components. Finally, in section 4, we briefly discuss the implications of our comparison results.

2. Software Agents vs. Software Components

Since the first release of the FIPA specifications in 1997 [9], researchers clearly understood the possibility of using agents as software components capable of exhibiting interesting characteristics, e.g., automatic reasoning and goal-directed behavior. In addition, FIPA chose to enable communication among agents by means of a CORBA interface, and this emphasized even more the strong interrelation between such abstractions. The long (and sometimes pointless) debate on the differences between agents and objects (see, e.g., [24]) originated from this comparison.

Component-oriented software engineering proposes extensions of objects, e.g., Web Services [25], JavaBeans [20], .NET components [8] and CORBA components [19], as a final answer to the need of reusable building blocks that can be assembled to realize complete systems. Such components are interoperable across networks and (possibly) languages and operating systems, to give a developer maximal freedom in the deployment of a system. Nevertheless, the long-pursued dream of component-oriented software engineering does not end with the realization of a technology for reusable units of software, but it considers also the following ideas:

1. Commercial Off-The-Shelf (COTS) components, i.e., components that are available in a public market and that are assembled to create a value-added system. The quality and the cost of the system basically depend on the quality and costs of every single COTS component. Market forces should help in decreasing costs while increasing the quality of available components.
2. Automatic assembly, i.e., the possibility of lowering the cost of the process of assembly of components through the use of automatic technologies. The quality and cost of the assembly process depend directly on the quality and cost of the available technologies for automatic assembly.

The use of COTS components combined with automatic assembly can lower the cost of a component-based system down to the direct investments related to each single component, summed to the cost of the technology for automatic assembly. Similarly, the quality of a system increases according to the quality of single components and of the technology for automatic assembly.

Our comparison between agents and components starts from Table 1, where we show some important aspects of components and associate them with their agentized counterparts. More precisely, we consider the most important features of the agents' metamodel and compare them with the corresponding features of the components' metamodel. In order to give maximal generality to our results, we avoid considering any particular metamodels, e.g., the Microsoft .NET metamodel for components or the SMART framework [15] for agents.

Table 1. Features of the agents' metamodel and their counterparts of the components' metamodel

Feature	Agent-oriented	Component-oriented
State	Mental attitudes	Attributes and relations
Communication	ACL	Metaobject protocol
Delegation of responsibility	Task and goal delegation	Task delegation
Interactions between parties	Capability descriptors	Interfaces
Interaction with the environment	New beliefs	Events

State Representation. Both agents and components are abstractions that comprise a state, but they have very different means to describe and to expose it to the outer world. The state of a component is represented though a set of attributes and a set of relations with other components. Attributes and relations can be public, i.e., other components can manipulate them directly. An agent has a mental state, i.e., its state is represented in terms of what it knows, e.g., its beliefs, and what it is currently pursuing, e.g., its intentions. The main differences between such models for representing the execution state are:

1. Agents cannot manipulate the state of another agent directly, but can affect it only through communication;

2. Agents have an explicit representation of their goals;
3. Agents have explicit knowledge of their environment, including other agents in the environment;
4. Except for a unique identifier, agents do not have public attributes.

One of the main advantages of the agents' approach is that agents can use general-purpose reasoning techniques to support deduction and means-end reasoning. On the contrary, the attributes and relations of a component's state are not structured in a logic framework, so it is difficult to use general-purpose techniques; any deduction and planning process must be coded explicitly in the methods of the component.

Communication. The main difference between agents and components is in the mechanism they use to communicate. Agents use declarative Agent Communication Languages (ACLs), while components use metaobject protocols [14]. In the agent-oriented approach, a message is sent only in an attempt to transfer part of the sender's mental state to the receiver. Let's take the FIPA ACL as an emblematic example: it defines *performatives*, i.e., semantic message types, together with *feasibility preconditions,* which must be true for the sender to send the message and *rational effects*, which are why the sender sent the message. When an agent receives a message, it can assert that the feasibility precondition holds for the sender and that the sender is trying to achieve the corresponding rational effect. This is basically a rather knotty way to let the receiver know that the sender wanted the receiver to know that the feasibility precondition holds for it and that it is actually bringing about the rational effect. The advantage of using a structured ACL, instead of a more natural exchange of representations of goals, is that it simplifies the development of reactive agents capable of complex interactions. Reactive agents with no reasoning capabilities can exploit the performatives of the ACL as triggers that activate the state machine of the underlying interaction protocol. This is what JADE [2] and similar platforms provide.

In the component-oriented approach, a message is sent for two reasons. The first is to directly manipulate the state of the receiver. This use of communications violates the autonomy of the component, which should be solely responsible for its own state. Most real-world technologies for implementing components prohibit direct manipulation of their states, in an attempt to satisfy a software engineering goal of minimizing this sort of coupling among components. The second reason for sending a message is to force the receiver to execute the body of a method for the sender without explicitly communicating to the receiver why it is being forced to do so. The responsibility for such an execution is completely that of the sender: it is responsible for guaranteeing that preconditions hold and for causing any changes in the rest of the system that might arise during the complete execution of the method.

Delegation of Responsibility. As we have just pointed out, both for agents and for components the delegation of responsibility is based on communication and the differences in the way they delegate responsibilities justify the differences in their communication models. In the component-oriented model, the sender is solely responsible for the possible outcomes of a message: it does not need to say anything more to the receiver than *"please do this under my responsibility."* Strictly speaking, components do not delegate responsibility to other components at all.

In the agent-oriented model, the receiver is solely responsible for the outcome of its own actions and the sender needs to say also why it is requesting the service. A very important communicative act that an agent can perform is delegating one of its goals to another agent, e.g., through the FIPA *achieve* performative. This special communicative act, known as goal delegation [7], is the basic mechanism that agents use to delegate responsibilities.

The components' metamodel does not comprise an abstraction of a goal and components can only use task delegation. Components achieve their (implicit) goals asking by forcing other components to perform actions; agents might achieve their (explicit) goals by delegating them to other agents. This is the reason why it is common to refer to the agent-oriented communication model as declarative message passing: agents can tell other agents *what* they would like for them to do without explicitly stating *how* to do it. On the contrary, imperative message passing is used for the component-oriented approach, because components cannot say to another component what to do without also saying how to do it.

The possibility of using only task delegation is a strong limitation for components, because goal delegation is a more general mechanism. First, task delegation is a special case of goal delegation: the delegated goal has the form *done(a)*, where *a* is an action, just like for the rational effect of the *request* performative in the FIPA ACL. Then, task delegation may inhibit optimizations. Consider, e.g., a component *S* with a goal *g* that needs component *R* to perform a_1 and a_2 to achieve it; *S* would ask to *R* to perform a_1 and then it would ask *R* to perform a_2. As the two requests are not coupled though the underlying idea that *S* is trying to achieve *g*, *R* cannot exploit any possible cross-optimization between a_1 and a_2.

If *S* and *R* were two agents instead of two components, *S* would simply delegate *g* to *R* and then *R* would decide autonomously the way to achieve it, i.e., it would decide how to perform a_1 and a_2. This approach couples a_1 and a_2 through *g*, thus enabling *R* to perform cross-optimizations between a_1 and a_2.

Interaction between Parties. The different communication models influence the way agents and components open themselves to the outer world. Components use interfaces to enumerate the services they provide and to tell clients how to get in contact with them. Sophisticated component models (see, e.g., [16]) equip interfaces with preconditions and postconditions.

The agent-oriented approach eliminates interfaces and provides agents with capability descriptors that depict what an agent can do, i.e., the possible outcomes of its actions, and how it can interact with other agents. The main difference between a capability descriptor and a postcondition is that the first can express how the state of the environment changes after the complete execution of an action. A postcondition can only assert how the state of the component changed after the action has been executed, because the environment is not part of the component's metamodel.

Interaction with the Environment. As we have just mentioned, an environment is a structural part of an agent's metamodel, while it is not part of a component's metamodel. Agents execute in an environment that they can use to acquire knowledge: agents are situated abstractions. Agents can measure the environment and they can receive events from it. In both cases, agents react to any change in the environment because of changes in their mental state. This is radically different from the component-oriented approach where the environment communicates with components only through reified events. Components can react to an event only by constructing a relation with a reification of the event itself.

The component-oriented approach seems to better respect encapsulation than the agent-oriented approach: the state of the component is changed only when the component itself decides to change it in reaction to an event. If we consider this in more detail, we see that the agent-oriented approach also respects encapsulation. Agents have reasoning capabilities that are ultimately responsible for any change in their mental state. Any direct push of knowledge from an agent's sensors to its mental state is ruled through reasoning, and the mental state remains encapsulated.

3. Semantically Reusing Agents and Components

Since the beginning of computer science, reusability has been considered one of the main properties of a development technology. First procedures, and later classes were a direct response to the need for creating reusable units of software to, e.g.,:

1. Speed-up the realization of new systems;
2. Ensure the quality of systems that are realized though the composition of a number of readymade units.

Component-oriented software engineering has already explored most of the peculiarities related to building a system in terms of assembled components, and it identified three concepts that any technology meant to improve reusability should take into account: semantic interoperability [11], semantic composability [18] and semantic extensibility [10]. No formal definition for such concepts is available in the literature and the general feeling is that formalizing such ideas would require concepts that are not part of the components' metamodel. On the contrary, we can give formal definitions of such ideas by taking into account very basic elements of the agents' metamodel.

Semantic Interoperability. Previous research on software components has explored the problem of semantic interoperability in many ways, and the agent community has also begun investigating the subject. For example, the recent work on the characterization of the capabilities of Web Services [16] follows the lines of established results (see, e.g., [12]). Strangely enough, there is no agreed upon definition for semantic interoperability and some variants of this concept are available in the literature with different names, even though this name has been in use for a while.

The idea of semantic interoperability comes from a reasonable extension of syntactic interoperability of components, i.e., the sort of interoperability that CORBA and "standards" with similar aims (e.g., DCOM and Java RMI) provide. CORBA allows components to exchange messages and provides an agreed upon syntax for such messages. The semantics of the exchanged messages is implicit, i.e., the semantics of a call to a method of a CORBA interface is implicitly defined as follows: the call to the method actually causes the execution of the body of the method. Nothing is said on the concrete outcome of the call, i.e., what would happen to the world outside of the component that executed the body of the method after such an execution would be completed. This outcome is considered application specific and relies completely on the programmer, who is responsible for reading the documentation of the interface for deciding when and how to call the method.

Syntactic interoperability inhibits automatic assembly of components, because a client has no means to reason about the effects of a call it might have decided to perform on one of the methods of a service-provider component. Semantic interoperability is about extending the interface of a component with an explicit formalization of the outcome of a method call in order to allow a client to decide autonomously when and how to invoke that method.

What we have just described can be applied to agents if we concur that invoking a method on a component is somehow similar to asking an agent to perform an action. Exploiting the characteristics of agents, we can formalize semantic interoperability as follows:

Definition (Semantic Interoperability, Client Standpoint) Given two agents C and $S \in \text{acquaintance}_C$, they are said to be semantically interoperable if and only if:

$$\forall g \in \text{goals}_C,\ (g \in \text{goals}_S) \in \text{goals}_C \Rightarrow [(g \in \text{goals}_S) \in \text{goals}_C] \in \text{knowledge}_S$$

where (note that all sets are time-variant, but time is not included in the notation for simplicity):

goals_A:	set of goals that A is pursuing;
knowledge_A:	set of propositions that A knows;
acquaintance_A:	set of components that A knows;

$\in$ denotes the set-element relationship.

This definition states that if (at some point in time) an agent C wants to achieve g, and it wants to delegate such a goal to S, then S will know of such a desire. In this way we can easily capture the lack of information loss, which is the core of semantic interoperability: if an agent has a goal, then it can transmit that goal to a service-provider agent without any loss of precision. It does not really matter how the goal is communicated, the only important result of the communication is the delegation of the goal to the service provider.

This definition of semantic interoperability takes the client standpoint, because C is the originator of g and nothing is said about S wanting to provide its services to a set of possible clients. A similar definition is trivially possible taking the server standpoint, but such a definition is basically equivalent to the one we showed and its discussion would not add much to the aims of this paper.

A fairly interesting consequence comes from this definition of semantic interoperability: if we consider a multiagent system where agents are only intended to interoperate semantically, then a very basic ACL with the *achieve* performative only is sufficient. This is not strange at all, because it easily generalizes the available work on ACLs, as discussed in the previous section.

Achieving semantic interoperability is not only a way for improving reusability, it is also a possible way for promoting optimization. With everyday syntactic interoperability, agents achieve their goals by possibly asking other agents to perform actions, i.e., exploiting task delegation towards other agents in an attempt to achieve their goals. Semantic interoperability exploits goal delegation and this may promote cross-optimization, as discussed briefly in the previous section.

Semantic Composability. The assembly of agents to realize a multiagent system is not only a matter of making agents communicate in the best way, but also allowing them to find each other. Interoperability is necessary, but not sufficient, for composability, Semantic interoperability requires $S \in \text{acquaintance}_C$ and we need to elaborate on this to achieve full semantic composability.

Semantic composability has been studied for a long time in the literature of component-oriented software engineering, starting from well-known results on the composability of objects obtained by researchers who are now active in the community of aspect-oriented programming [1]. The basic idea behind semantic composability is that a component should be free to compose the services provided by a set of service-provider components with no constraints deriving from locating the right service providers or from possible mismatches between the interfaces of such service providers. It requires that the things being composed not only have compatible interfaces, but also make consistent assumptions about the world.

Semantic composability has already been extended to agents [21], and we can make it more formal by exploiting the same technique that we used for semantic interoperability. We can say that two agents are semantically composable if no constraint is imposed on the way agents delegate goals and, more formally, we can define semantic composability as follows:

Definition (Semantic Composability) Given a set of n agents $MAS=\{A_1, A_2, \ldots, A_n\}$, they are said to be semantically composable if and only if:

$$\forall C \in MAS,\ \forall g \in \text{goals}_C,\ \exists S \in MAS : (g \in \text{solves}_S) \Rightarrow [(g \in \text{goals}_S) \in \text{goals}_C] \in \text{knowledge}_S$$

where (note that all sets are time-variant, but time is not included in the notation for simplicity):

solves_A:	set of goals that A can solve.

This definition states that if an agent *C* has a goal and there is an agent *S* available in the multiagent system capable of achieving such a goal, then *C* can delegate the goal to *S* with no loss of precision caused by communication. In this way we can capture the lack of information loss that semantic interoperability entails, without the need of requiring *C* to know *S* and to desire to delegate its goal to that *S*. It does not really matter how or to whom the goal is communicated, the ultimate result of the composition is that an agent of the multiagent system would achieve the goal for *C*.

This definition does not require the client to know the service-provider agent prior to the delegation, and it does not guarantee that the chosen service provider would be known after the delegation. This is compatible with the common approach of explicitly choosing the service provider, because the two approaches are both captured by the definition: the client can identify the service provider of choice in its goal. For example, if an agent *C* wants *S* to achieve goal *k* for it, then the goal that *C* is bringing about is actually $g=[(k \in \text{goals}_S) \in \text{goals}_C] \in \text{knowledge}_S$.

Semantic Extensibility. Taking the literature on component-oriented software engineering into account, we see that reusability is pursued not only by means of composing reusable components, but also by making such components extensible [5]. Extensibility provides mainly two possibilities of reuse:

1. Implementation of new components as extensions of available components;
2. Substitution of an existing component with a different one with (possibly) no changes in the rest of the system.

The first approach is traditionally considered the base of object-oriented programming: it supports the creation of new classes of objects by means of inheritance and polymorphism. This is still a good way to bring about reusability, but nowadays the second approach is preferred because it allows reusing entire systems and not only single classes. This so-called framework-based reusability relies on the possibility of substituting a component with another component without the rest of the system (i.e., the framework) being aware of such a substitution.

Object-oriented and component-oriented paradigms achieve framework-based reusability by means of inheritance and polymorphism, because they assume that if two components belong to the same class, i.e., they are of the same type, then they are substitutable. This is obviously not enough and some extensions to such an approach have been already proposed [17]. In particular, the main problem of approximating substitutability with type equivalence is that two classes may provide the same methods, but the semantics of such methods, i.e., what they do on the world outside of the component, may be completely different. In other words, two classes may be structurally identical, but semantically different [10].

The idea behind semantic extensibility is that we want to have the possibility of substituting a component with another component extending the features provided by the first component, while preserving the semantics of the operations that clients were able to perform before the substitution.

Taking the agent-oriented mindset and exploiting the formalisms that we have introduced previously in this section, we can formally define semantic extensibility as follows:

Definition (Semantic Extensibility) Given two agents *B* and *D*, we can say that *D* is a semantic extension of *B* if and only if:

$$\text{solves}_B \subseteq \text{solves}_D$$

This definition states that (at each point in time), what *B* can solve is also solved by *D*, i.e., from the point of view of any possible client interested in the services that *B* may provide, they are substitutable.

Semantic extensibility together with semantic composability maximizes the reusability of agents, at least if we adopt the assumption of considering agents as the atomic units of reuse. Agents are composed freely on the basis of their goals and they can be substituted with other agents with extended capabilities with a complete reuse of the multiagent system surrounding the substituted agents.

Approximating Semantic Reusability. The model of reusability that we have just discussed is obviously idealistic, because it does not provide any operational means for supporting composability and extensibility. Nevertheless, if we make the sets goals_A and solves_A public and explicit, then semantic composability is just a matter of passing a goal from a client to a service provider and, in the most general case, it is just a matter of communication. We could exploit a matchmaker agent capable of connecting a client with a service provider, or we might rely on the middleware infrastructure. In this last case, e.g., we could exploit a tuple space forwarding goals from clients to service providers, or we could rely on a direct message passing that the programmer coded explicitly in the program of the client.

Similarly, making solves_A public and explicit guarantees semantic extensibility, because a client can always check solves_S before requesting a service from a service provider *S*.

Unfortunately, goals_A and solves_A cannot be computed in the most general case and we can only rely on public and explicit approximations of such sets. Components and agents provide different approximations of such sets and the advantages that agents have over components in terms of reusability derive from these different approximations.

The ParADE framework [3] was designed to maximize the interoperability of agents and it approximates semantic reusability as follows [4]:

1. beliefs$_A$ ≈ knowledge$_A$: the knowledge of an agent is approximated with what the agent believes, i.e., what it has deduced by applying steady rules to its measurements of the environment;
2. intentions$_A$ ≈ goals$_A$: the set of goals of an agent is approximated with the set of intentions that it calculates from its beliefs and from the rules that drive its planning engine;
3. capabilities$_A$ ≈ solves$_A$: the set of goals an agent can solve is approximated with the set of post-conditions of its feasible actions. These postconditions take into account the state of the agent and of the environment after the complete execution of an action.

The components' metamodel relies on even stronger assumptions:

1. state$_A$ ≈ knowledge$_A$: the knowledge of the component is approximated with the state of the component, i.e., the values of its attributes and its relationships with other components;
2. postcondition-of-next-call$_A$ ≈ goals$_A$: the set of goals of a component is approximated with a singleton set that contains the postcondition of the method that the component is about to invoke;
3. postconditions$_A$ ≈ solves$_A$: the set of goals a component can solve is approximated with the set of postconditions of its methods. Such postconditions are defined on the state of the component after the complete execution of a method and nothing is said about the state of the environment.

Roughly, agents approximate semantic reusability better than components, because the element of the architecture that is in charge of enabling the flow of information between a client and a service provider, e.g., the matchmaker agent, has more precise information to perform its job.

Agents approximate semantic extensibility better than components because the capability descriptors that they use comprise conditions on the environment surrounding the agents, while the postconditions of the methods of components consider only the state of a service provider after the complete execution of a method. Therefore, agents can give very precise information on the outcome of an action for the purpose of guaranteeing semantic extensibility.

4. Discussion

Agents not only are suited for uncommon types of applications where the advanced characteristics of agents, such as learning and autonomy, are required, but also represent a valid alternative to other solid technologies because agents:

1. Provide the developer with higher level abstractions than any other technology available today [3];
2. Have concrete advantages over components in terms of reusability.

The first point, i.e., working with higher level abstractions, has well-known advantages, but it also has a common drawback: slower speed of execution. In order to fully exploit the possibilities of agents, we need to implement an agent model with some reasoning capabilities and agents of this sort are likely to be slow. Nowadays, this does not seem a blocking issue because speed is not always the topmost priority, e.g., time-to-market and overall quality are often more important.

As far as the second point, reusability, is concerned, the improvement that agents obtain comes at a cost: slower speed again. The use of goal delegation instead of task delegation requires, by definition, means-end reasoning and we face the reasonable possibility of implementing slow agents.

Fortunately, in both cases the performances of agents degrade gracefully. We can choose how much reasoning, i.e., how much loss of speed, we want for each and every agent. In particular, we may use reasoning for agents that:

1. Are particularly complex and could benefit from higher level abstractions;
2. We want to extend and compose freely in many different projects.

On the contrary, we can rely on reactive agents, or components, when we have an urge for speed. This decision criterion seems sound, because the more complex and value-added an agent is, the more we want to reuse it and compose it with other agents. Moreover, reactive agents are perfectly equivalent to components and we do not loose anything using the agent-oriented approach instead of the component-oriented approach.

References

1. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: Abstracting Object-Interactions using Composition-Filters, in Guerraoui, R., Nierstrasz, O., Riveill, M. (Eds), Object-based Distributed Processing, Springer-Verlag (1993)
2. Bellifemine, F., Poggi, A., Rimassa, G.: Developing Multi-agent Systems with a FIPA-Compliant Agent Framework. Software Practice and Experience 31 (2001) 103–128
3. Bergenti, F.: A Discussion of Two Major Benefits of Using Agents in Software Development, Engineering Societies in the Agents World III, LNAI 2577, (2003) 1–12
4. Bergenti, F., Poggi, A.: A Development Toolkit to Realize Autonomous and Inter-operable Agents. Procs. 5th Int’l Conference on Autonomous Agents, (2001) 632–639
5. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley (1994)
6. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Modeling Early Requirements in Tropos: A Transformation Based Approach. 2nd Int’l Workshop on Agent-Oriented Software Engineering (AOSE-2001), Montreal, Canada (2001).

7. Castelfranchi, C.: Modelling Social Action for AI Agents. Artificial Intelligence 103(1) (1998)

8. European Computer Manufacturer's Association: Standard ECMA-335, Partition II Meta-data Definition and Semantics. Available at http://www.ecma.ch

9. Foundation for Intelligent Physical Agents. Specifications. Available at http://www.fipa.org

10. Fankhauser, P., Kracker, M., Neuhold, E. J.: Semantic vs. Structural Resemblance of Classes. ACM SIGMOD RECORD 20(4) (1991) 59–63

11. Heiler, S.: Semantic Interoperability. ACM Computing Surveys, 27(2) (1995) 271-273

12. Jeng, J-J., Cheng, B. H. C.: Specification Matching for Software Reuse: A Foundation, in Procs. ACM SIGSOFT Symposium Software Reusability, ACM Software Engineering Note, Aug. 1995.

13. Jennings, N. R.: On Agent-Based Software Engineering. Artificial Intelligence, 117 (2000) 277–296

14. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)

15. Luck, M., d'Inverno, M.: A Conceptual Framework for Agent Definition and Development. The Computer J. 44(1) (2001) 1–20

16. McIlraith, S., Martin, D.: Bringing Semantics to Web Services, IEEE Intelligent Systems, 18(1) (2003) 90–93

17. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall (1997)

18. Pratt, D. R., Ragusa L. C., von der Lippe, S.: Composability as an Architeture Driver. Interservice/Industry Training, Simulation and Education Conference, Orlando, Florida (1999)

19. Suhail, A.: CORBA Programming Unleashed. Sams (1998)

20. Sun Microsystems: JavaBeans Specification: Version 1.1. Available at http://java.sun.com

21. Sycara, K., Widoff, S., Klusch, M., Lu, J.: LARKS: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. In Autonomous Agents and Multi-Agent Systems, 5, (2002) 173–203

22. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison Wesley (1998)

23. Werken Company. The drools Guide. Available at http://www.drools.org (2003)

24. Wooldridge, M. J.: Intelligent Agents. In Weiss, G. (Ed) Multiagent Systems, MIT Press (2000) 27–78

25. World Wide Web Consortium: Web Services Description Language (WSDL) 1.1. Available at: http://www.w3c.org