PAPER

Towards clarifying the importance of interactions in Agent-Oriented Software Engineering

Joaquin Peña

The Distributed Group
University of Seville
Avda. de la Reina Mercedes, s/n.
Sevilla 41.012 (Spain)
{joaquinp}@tdg.lsi.us.es

Renato Levy

Intelligent Automation Inc.
15400 Calhoun Drive
Suite 400, Rockville
MD 20855, USA
{rlevy}@i-a-i.com

Rafael Corchuelo

The Distributed Group
University of Seville
Avda. de la Reina Mercedes, s/n.
Sevilla 41.012 (Spain)
{corchu}@tdg.lsi.us.es

Abstract

Interactions between subparts of a system have been recognized as the source of complexity in many fields ranging from physics, sociology, neurology, to software engineering. Agent-Oriented Software Engineering (AOSE) was born under the promise of conquering complexity and enabling the development of more complex software. However, current AOSE approaches do not provide enough engineering tools to deal with the complexity derived from interactions. More mature fields such as economy or component-based software systems have recognized that interactions present a predominant role in the determination of the desired outcome providing mature background that can be applied to AOSE.

AOSE may improve its ability to deal with complex systems by improving the tools applied to manage agent's interactions in the overall design of the system. In this paper, we justify this assessment and propose some principles to improve AOSE methodologies regarding complexity.

keywords: Complex MultiAgent Systems, organizational modeling, interaction taxonomy.

1 Introduction

The importance of interactions can be supported over two main facts: i) interactions have been presented as the main source of complexity by many authors, and (ii) the interaction is the central abstraction in many mature fields.

These facts indicate that we should pay special attention to interactions when modeling a Multi-Agent System (MAS). This focus is possible based on the premise, accepted in the enterprise organization field [12], and at later by the agent field, that an organization can be observed from two different points of view: functional/interaction

and structural. Roughly speaking, the model of the functional organization is built of roles and interactions while the model of the structural organization is built of agents and interactions.

Despite of their close relationship, both types of organization views can be modeled independently. This fact allows the designers to model the interaction process ignoring the organizational structure until it is clearly understood how it operates. This modeling process reduces the complexity of models to be managed at first stages of the software process and eases the comprehension of complex behaviors.

In addition, interactions do not present always the same level of complexity. Consequently, their modeling and implementation requires different strategies depending on their complexity nature.

The main contributions of this paper are to show the importance of interactions, provide a taxonomy that classifies interactions based on their complexity, and outline some general guidelines for facilitating the modeling process of Multi-Agent Systems (MAS) that present different levels of complexity. As a result, we also propose future research lines in the field.

This paper is organized as follows: Section 2 presents the related work that support the importance of interactions regarding structure/architecture; Section 3 presents our taxonomy of interactions; Section 4 shows the facts that allows the separation of interactions and structure; Section 5 shows the main principles to manage interaction complexity; Finally, Section 6 presents our main conclusions and outlines some future research lines.

2 The importance of interactions

The importance of interactions has been already established in the agent literature as well as in several other fields.

2.1 Interactions in other fields

Some advanced Object-Oriented Software Engineering approaches, or even in the traditional sociology field, already present a predominant role regarding structural features to interactions to the point in which all the modeling process is focused on them. OOram [20] is a good example of an Object-Oriented approach where the whole development cycle is focused on interactions. OOram's authors state that the main advantages of focusing on interactions is the improvement of reuse, traceability and the ability to cope with complexity [19].

Furthermore, in sociology, interactions have been emphasized by important authors such as the German sociologist Max Weber. Weber in his concept of *ideal bureaucracy* emphasizes the form, or in other words, the interrelationships between the members of an organization. Reenskaug in 1988, the author of OOram, states in [19] that object-orientation was born by the hand of Weber, as an argument for basing on his ideas, and concluded that OO-methodologies must focus on interactions.

In addition, this fact is also ratified by the research done in other mature fields: i) in the component world, Syzpersky and D'Souza also emphasizes the importance of focusing on interaction instead of architecture (structure in MAS) in complex systems [22, pag. 124][4]; ii) In the distributed field, several authors has also claimed to approaches that focus on interactions, i.e. Francez and Forman who claim for the importance of modeling complex interactions as a singleton and who also works on functional groups of interacting elements [7]; iii) the last version of UML provides also modeling artifacts to perform interaction-centered modeling emphasizing and improving the role concept regarding previous versions [14, 18].

2.2 Interactions as the source of complexity in AOSE

Several authors agree that the complexity of MASs is a consequence of their interactions [9, 13]: *Complexity is caused by the collective behavior of many basic interacting agents.* In fact, many authors point out that the complexity of MASs is the consequence of those interactions among agents, and that these interactions can vary at execution time, and cannot be predicted thoroughly at design time, namely, emergent behavior. The reasons for the emergence can be traced to two features present in MASs: self-adaptation, and self-organization [8, page 20–21][9, 13]. It is important to observe that this

capability of demonstrating emergent behavior is the key factor that drove us to implement MAS solutions in the first place, since this key capability is essential to address solutions to the targeted domains.

3 Characterizing complex interactions

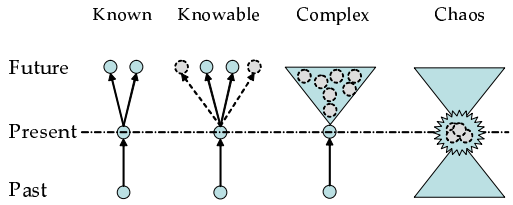

Figure 1. Complexity and Predictability

Although in Section 2.2, we show that interactions are seen as the main source of complexity in MASs, a large MAS is usually composed of many parts which do not present the same features. Some parts of a MAS could be fully predictable not presenting any emergent feature, while some other parts of the same MAS could be highly complex presenting a high-degree of self-adaptation and self-organization. In the field of enterprise organization, Snowden and Kurtz recognize this fact [21]. These authors divide an organization into the following domains whose main features are summarized in Figure 1:

1) Ordered Domain: Stable cause and effect relationship exist. In this domain, the sequence of events/actions of the organization can be established as a cause/effect chain. It represents the predictable part of the system. This domain is further divided into:

1.1) Known Domain: [Figure 1] In this domain, every relationship between cause and effect is known. The part of a MAS in this domain is clearly predictable and can be easily modeled.

1.2) Knowable Domain: This is the domain that while stable cause and effect relationships exist, they may not be fully known. In general, relationships are separated over time and space in chains that are difficult to fully understand. The key issue is whether, or not, we can afford the time and resources to move from the knowable to the known domain. In Figure 1 this is represented by a higher number of future directions given a certain present state.

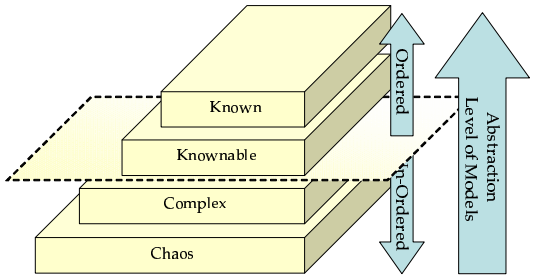

Figure 2. Domain of a problem depending on the abstraction level of models

2) Un-ordered: This domain presents unstable cause and effect relationships between interactions in the system. It represents the unpredictable part of the system. This domain is divided into:

2.1) Complex Domain: There are cause and effect relationships between the agents, but both the number of agents and the number of relationships defy categorization or analytic techniques. Unfortunately, relationships between cause and effect exist but they are not predictable. This domain presents retrospective coherence. That is to say, coherence can be only established by analyzing past history of the system. Unfortunately, future directions, although coherent, cannot be predicted. In Figure 1 the past events/actions can be understood as a single chain of cause/effects, but when we try to extrapolate and predict future changes, the solution space is too wide to be analyzed.

2.2) Chaos Domain: There are no perceivable relationships between cause and effect, and the system is turbulent; we do not have the response time to investigate change. Despite some previous work in this area, chaotic domains are still out of reach from the point of control theory. Agents systems have been used to model such domains, but strictly limited to simulation.

The complexity characterization proposed by Snowden and Karts cannot fully describe a MAS from a software engineering perspective. In an AOSE approach, complexity depends on the abstraction level in which we are working. Thus, software models can be highly complex while requirement models may be quite simple. In order to fully describe the MAS from the point of view of this paper, we must introduce another dimension: the level of abstraction of models. Figure 2 shows this dependency.

When studying a certain problem, we may need more or less details of it. That is to say, there exists a certain level of abstraction that provides only the level of detail needed for the software phase and the problem at hand. The level of abstraction is an important fact in the categorization of the MAS. A MAS categorization may vary anywhere from the known domain down to the chaos domain depending on the level of abstraction at which the MAS is observed and depending on its features.

Similarly, the complexity level of an interaction depends on the level of abstraction in which its features regarding emergence are observed. This principle can be visualized by the following interaction categorization in Figure 3.

The complexity of an interaction, or set of interactions, depends on their nature and on the effort taken in understanding its details, such as, their predictability and flexibility, and their level of abstraction. Our proposed interaction categorization is based in the space defined by these two axes. Figure 3 shows the classification of interactions in three categories: known, knowable, and complex interactions.

Known interactions are the less flexible, they do not present emergence, and all their details can be identified. Complex interactions present a higher degree of flexibility and can only be described with higher level patterns emphasizing most important details. Knowable interactions represent a middle point between both of them.

In addition, agents undertaking complex interactions may present a high degree of autonomy, proactivity, reactivity and, obviously, social abilities. The further a subpart of an agent system moves from known into complex interactions the further its abilities, as described above, are intensified. We must observe that the need to describe (and generate) complex behavior from simpler constructs was the reason that drove us to agent based systems in the first place, therefore our goal, must be to describe the system as it is perceived (complex), and increase details until the desired behaviors can be synthesized.

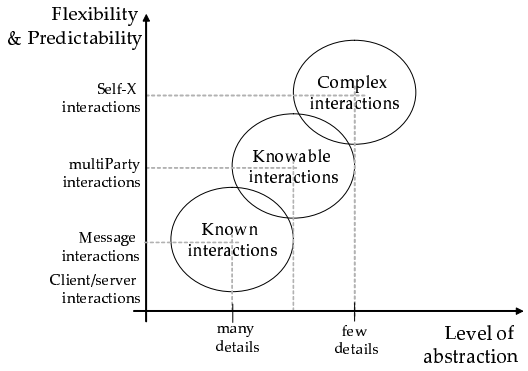

Figure 3. Proposed interactions taxonomy regarding complexity

4 Isolating interactions

No agent is an island. Since agents are limited to some environment and have limited abilities, complex problems are usually solved by a set of agents [3]. Hence, an organization represents a group of agents formed in the system in order to get benefits from one to another in a collaborative or competitive manner.

Therefore, a sub-organization emerges only when some kind of interaction between its participants exists, either through direct communication by means of speech acts or through the environment. The structure of an organization is underlined by the nature of their interactions; hence it is vital to clearly understand the interactions within a MAS system in order to determine its sub-organizations.

Other authors [2, 23, 6] have recognized this fact and proposed to see the organization of MASs from two different points of views:

The interaction point of view: it describes the organization by the set of interactions between its roles. The interaction view corresponds to the functional point of view.

The structural point of view: it describes the agents of the system and how they are distributed into sub-organizations, groups,

and teams. In this view, agents are also presented into hierarchical structures showing the social architecture of the system.

The former is called *Acquaintance Organization*, and the later is called *Structural Organization*. Both views are intimately related, but they show the organization from radically different points of view.

Since any structural organization must include interactions between their agents in order to function, it is safe to say that the acquaintance organization is always contained in the structural organization. Therefore, if we determine first the acquaintance organization, and we define the constraints required for the structural organization, a natural map is formed between the acquaintance organization and the correspondent structural organization. This is the process of assigning roles to agents [23]. Thus, we can conclude that any acquaintance organization can be modeled orthogonally to its structural organization [11].

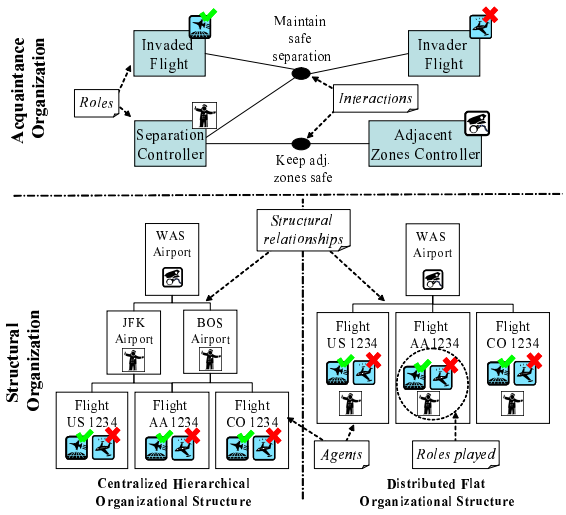

Figure 4. Acquaintance *vs.* Structural Organization: US air traffic control example

Assuming the United States air traffic control as an example, the functionality is to allow the planes to fly, guiding them safely through airport to airport.

Safety requirements force the air separation between flights to be kept at all times. The actual interaction in this case is between two flying airplanes in order to keep at least a minimum distance. At the acquaintance organization, shown in the top of Figure 4, these planes must be observed from the role they play in an invasion of air space, that is to say, roles *invader* and *invaded* in the figure. In addition, two controllers are present: one for controlling safety inside a zone (*Separation Controller*), and another for controlling safety between adjacent zones (*Adjacent Zones Controller*).

The structural constraints of the system demand a hierarchical structure, in which one *Separation Controller* would observe all planes in a given area and keep them safely apart. Furthermore, since many of such areas exist and planes close to the borders of the area may violate the airspace of adjacent areas, interactions between *Separation Controller* to *Adjacent Zones Controller* were also needed. These constraints and the natural consideration on lives and liabilities evolved into a highly hierarchical structure. Despite the fact that such structures decrease flexibility in procedures and enforce mistake free organizations [12]. The result is a system with several layers of interaction being the ones between the controllers and pilots the base of the pyramid as it is shown in the bottom-left corner of the figure.

The advent of GPS navigational systems and other modern communication technologies have changed the constraint map. Now, planes know exactly where they are and can have a perception of where its peers are. This allows the separation responsibility of the system to be directed to the interaction key player, namely, the flight itself, leaving for the controller a less strenuous task, of monitoring which was already present at the acquaintance level. We must observe that in both cases the acquaintance organization is the same, the target interaction objective is still the aircraft separation, but with the change of constraints, a complete different structural organization is possible, e.g., such shown on the bottom-right corner of the figure, where the role controller is played by airplanes.

In addition, starting the analysis with a certain organizational structure in mind (by means of agents), even if based on a real organization, it will drive the deployment of the MAS. Consequently, the initial subdivision in interactions and roles may not be optimal.

Real life organizations are known to present less then optimal structures. The presence of such organizational mistakes has been well studied in economics [12], hence the field of operational research. Using the real life organization as the

initial drive for the MAS system without further consideration will mimic its mistakes and may lead to some important misconstructions in terms of agent systems. Some of the common errors that can be induced are: agents coordinated by more than one agent, agents introduced to cover the relations between several sub-organizations, redundant agents with the same profile placed in different sub-organizations, etc.

As we show later, since interactions are the main source of complexity, we should not bother about organization structure at initial analysis. This approach facilitates the process of understanding the complex behavior of a MAS and minimize structural mistakes. Thus, when we consider the relationship between real organizations and their constraints into the system architecture; we must abstract the organization and let it be modeled by means of roles and interactions during the analysis phase. Later, these roles can be mapped into the concrete agents and structured as the real organization trying to fit the real life organization and trying to minimize structural mistakes.

Interactions and role-to-role relationship are therefore the primary concept of the engineering process of MASs and structural organization rises because of them.

5 Managing complex and knowable interactions

In this paper we will not cover known and chaotic interactions since the former can be easily modeled and implemented. Chaotic interactions are the building blocks of systems whose behavior are uncontrollable using only classic control theory. Truly chaotic systems, as existent in nature, can be seen as the higher order form of the complex system. Once we can model a chaotic system in attractors, such as the ones used in advanced control theory, the same principles used to compose complex interactions can be used to compose the interactions of the targeted chaotic system. This level of Chaos theory is beyond the scope of this paper.

In order to model and implement complex and knowable interactions several conceptual tools can be used. In [9], Jennings adapts the three main principles to manage complexity in traditional Software Engineering, as proposed by Booch in [1]: Abstraction, Decomposition and Organization/Hierarchy. Reuse and Automation are also two key powerful tools that help in modeling and implementing this sort of interactions.

5.1 Abstraction

Abstraction consists on defining simplified models of the systems that emphasizes some details avoiding others. The power of abstraction comes from limiting the designer scope of interest, allowing the attention to be focused on the most important details. Abstraction can be applied to interactions that fall in the complex and knowable domains, enabling us to abstract from how emergence can be obtained until the designer is ready to address the issue.

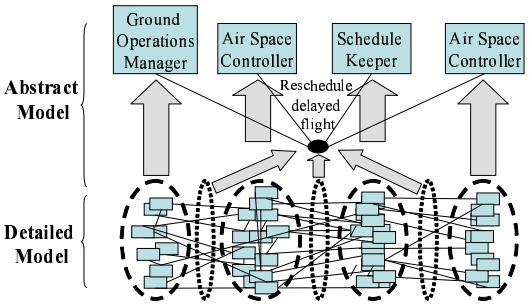

Figure 5. Abstraction Example

In the air traffic example of figure 5, the airport can be seen as a set of roles. The *Ground Operation Manager* group of roles manage resources required to adapt to the delays, i.e. change of gate. The *Schedule Keeper* group of roles need to guarantee that the new schedule is feasible and keep the schedule current, *Air Space Controller* group of roles adjust airspace routes and are responsible of take off and landing. Finally, the last group of roles involved, *Cost Quantifier*, are responsible of quantifying and prioritizing the set of available reschedules. Thus, the actual airport behavior is a composed effect of all these roles executing concurrently. The final behavior is extremely complex, but each group of roles can be defined well as a single abstract role within the known domain. Once the details of the airport are abstracted; its own internal make-up becomes a feature of the system.

The same technique can be use at the interaction design level. In the air traffic domain a simple interaction in which a flight is delayed on its arrival, may spawn a series of interactions which

are required to re-schedule all possible flights and airports impacted by the original delay. However, if we observe this set of interactions abstractedly from the acquaintance organization point of view, the problem can be reduced to a single interaction between roles with active control over the schedule, *reschedule delayed flight* abstract interaction as shown in figure 6. The actual airports and flights and resources involved in the problem, become less relevant, and can be re-introduced in the picture, later in the design process.

5.2 Decomposition

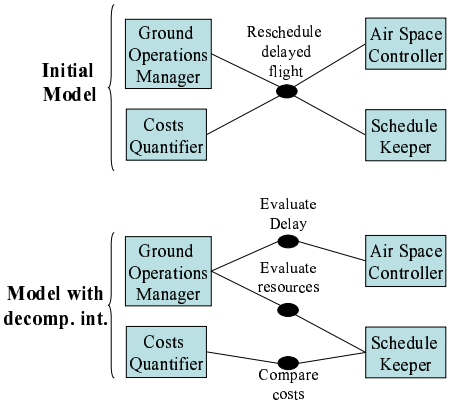

Figure 6. Decomposition Example

Decomposition consists on the principle "divide and conquer". It helps to limit the designer scope to a portion of the problem. Regarding interactions, it may help to decompose complex and knowable interactions into finer grain interactions. These finer grain interactions can be added with details which can not be applied when more abstract interactions are managed. In addition, using decomposition the interactions obtained can be implemented with less effort.

Continuing with the practical example given before, we depict in Figure 6 how the complex interaction of re-scheduling the outgoing flights due to a delay of an incoming flight can be decomposed by the designer on several simpler interactions. For example, we can decompose it on interactions such as: assessment of delay impact of outgoing flight in their own destinations (*Evaluate delay* interaction in the figure), quantification and prioritizing the assessments (*Compare costs* interaction in the figure), evaluation of optional resources (*Evaluate resources* interaction in the figure), etc.

In addition, a model that becomes too large and complex can be also decomposed into several models. This allows each sub-problem to be studied in isolation, ignoring the complexity derived from the interactions between sub-problems. For example, the model that shows how delays are dealt with in the system could be decomposed by designers into three models: one for modeling the occurrence of the delay, another for showing how the conflicts spread out, and a third one that determines flights re-scheduling. Thanks to this decomposition, the rescheduling problem for example can be refined easier without taking into account the reason of the reschedule.

5.3 Composition

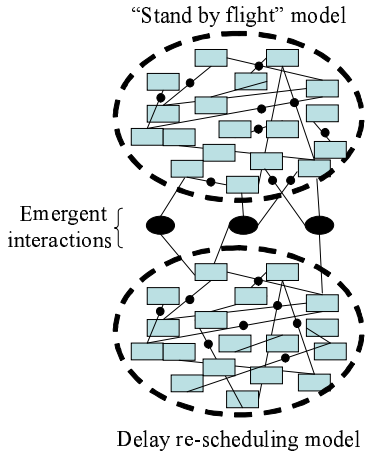

Figure 7. Composition Example

Composition consists on identifying and managing the inter-relationships between the various subsystems in the problem. It makes possible to group together various basic components and treat them as higher-level units of analysis. Composition makes it possible to describe the high-level relationships between several units. Composition helps to discover subtle interactions between several sub-organizations of the MAS.

In a sense, composition is the required mechanism in order to recreate the abstract complex interactions from their simpler components. In addition, the composition of acquaintance in a sub-organization can be used as the means to build the structural organization. As the components of an agent are fused, we can draw a "black box" an overlook its internals based only on the interfaces (roles) that cross the boundaries of the box.

This process will lead to view a group of agents as an agent on itself, and help build the hierarchical structure of the organization.

A composition example can be observed in the air traffic domain (see Figure 7), by considering the mechanism that allows "stand by" flight. In "stand by flight", a passenger is able to transfer a ticket to another flight between the same origin and destination, within the same airline, if room is available. This mechanism is independent from the delay re-scheduling problem. In this situation, the designer can compose both models to discover the new interactions that interrelates both systems, allowing all passengers from an impacted flight to be re-scheduled into other flights.

The advantage of modeling both problems in isolation abstracts these interactions and makes the modeling process easier. In addition, it improves the reuse of models, since their interdependencies would limit the reuse of a combined solution only into systems where both conditions occur.

The same principle applies into role composition. Roles are artifacts that can be combined. These artifacts may result on composed roles, or agents playing several roles. When agents are defined as a result of composition, the definition of a structural organization stars to be formed.

5.4 Reuse

Reuse is based on using previous knowledge in designing MASs. Reuse may save significant time and effort for modelers, allowing them to customize parts of the system instead of constructing them from scratch. This process has been the holy grail of software engineering since its inception and agents rather then facilitating the process make it even more essential. The benefits of reuse are many, such as errors avoidance, reduced maintenance, and faster development of highly complex models.

Reuse involves process, modeling artifacts, techniques, guidelines, and models of previous projects [4, 10]. For example, complex interactions that require negotiation or self-adaptation algorithms may be directly implemented by reusing these algorithms. In addition, the reuse of an interaction/algorithm provides encapsulation which also abstracts their inner complexity.

If code reuse is an issue, we can compose models to reach the macro level requirements of the system from smaller behavior components. Most requirements will only deal with the macro-level behavior; the abstract models generated from such requirements must be incrementally built until an implementable model is reached. If components that implement parts of the behavior are already available, then the designer may introduce their abstract models within the analysis.

It is important to observe that the behavior of the components used, may not have been originally meant to be applied together. The automatic composition of the reusable components may lead to strange and unanticipated behavior. Hence, we must compose them manually, as showed in the composition principle, searching for the set of emergent interactions and the changes that relate them properly.

An example of a CASE tool that enforces the reuse model is the Design of Interactions and Verification of Agents, known as DIVA. DIVA enforces the reusability by applying the composition method known as Three-Tier-Architecture [5].

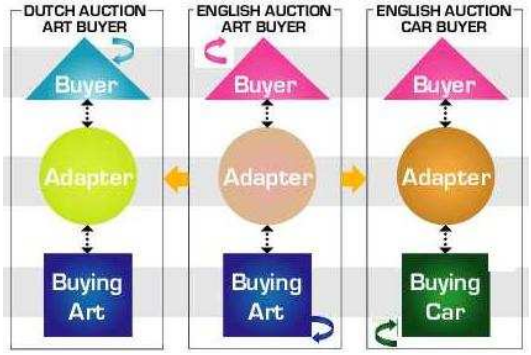

Figure 8. Three-tier architecture - reuse example

As it is shown in Figure 8, in the Three-Tier-Architecture roles are decomposed in syntax, context and glue code. The syntax component is responsible to enforce the expected communication and flow of the interaction, regardless of the domain in which this interaction occurs. For example a Dutch auction interaction process is invariant regardless of the product being negotiated. Since the syntax tier is context free it can be formally verified that it in fact achieve its goals, reducing debugging and development time.

As we would expect, the participant agents implementing the role must be able to understand the domain. This is the task of the context tier. The

context tier may be standard domain algorithms and even legacy code. Since syntax and context tiers are built independently, the glue tier is necessary to translate and adjust their interfaces.

The Three-Tier-Architecture accomplishes reusability at two levels. In the first level, syntax tiers that implement the protocol followed by its role can be reused across domains. The second level is the reuse of domain dependent libraries or algorithms as components of the context tier.

5.5 Automation

The use of tools such as DIVA to enforce architectures rich in reuse, underlines the importance of the last of the five principles presented above. Automation consists on automating the composition, decomposition and testing of models. CASE tools such as DIVA, organize the existent array of reusable code, and allow humans to explore and quickly compose/decompose variations on the system under design, reducing the development cycle significantly. Automation of testing, such as the formal verification of the syntax tier, can help us to uncover unexpected situations or interactions that prevent the participants of the interaction to achieve a viable final state.

6 Conclusions

We have shown the importance of interactions and we have outlined how complexity derived from interactions can be managed from an engineering perspective. Figure 9 summarizes how abstraction, decomposition and composition can be applied to transit between known, knowable and complex interactions. Abstraction is lower for known interactions since their details can be easily modeled, while the level of abstraction required for complex interactions is higher, since they can be only understood when observed by their most important features. In addition, complex interactions modeled abstractedly can be transformed into knowable and known interactions by means of decomposition. In the reverse process, known interactions, such as those found in code models, can be transformed into knowable and complex interactions by means of composition. The composition process will uncover emergent behaviors inherent to its internal components. The final resulting complex interaction can be further abstracted.

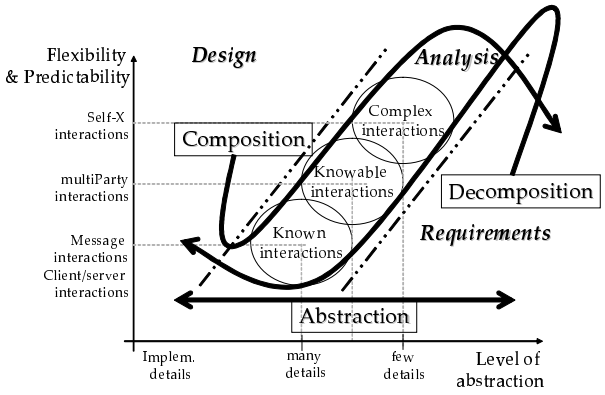

Figure 9. Usage of conceptual tools to manage taxonomy complexity of interactions

This paper opens us the path to a set of future research lines. We have started working on several of them while some others are still open.

We must explore the modeling artifacts to represents the acquaintance organization. In Ref. [16] and in DIVA, we present and apply our first results on this topic. Abstract programming primitives for representing the interactions between several agents abstractedly are also quite valuable for decreasing the complexity of code of MASs. In this aspect, new design and architectural patterns, such as the three-tier-architecture, should be also studied. Automatic or semi-automatic techniques to decompose and compose interactions can also be rather valuable to delegate the effort of modelling complex interactions to a CASE tool. In Refs. [15, 17], we present our first approaches toward automate the composition and decomposition of acquaintance models. Finally, automatic or semi-automatic techniques to map the acquaintance organisation over the structural organisation are also quite valuable to manage the complexity derived from interactions.

References

[1] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1990.

[2] G. Caire, F. Leal, P. Chainho, R. Evans, F. Garijo, J. Gomez, J. Pavon, P. Kearney, J. Stark, and P. Massonet. Agent oriented

analysis using MESSAGE/UML. In *Proceedings of Agent-Oriented Software Engineering (AOSE'01)*, pages 101–108, Montreal, 2001.

[3] C. Castelfranchi. Founding agent's "autonomy" on dependence theory. In *14th European Conference on Artificial Intelligence*, pages 353–357. IOSPress, 2000.

[4] D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison–Wesley, Reading, Mass., 1999.

[5] Kutluhan Erol, Jun Lang, and Renato Levy. Designing agents from reusable components. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 76–77. ACM Press, 2000.

[6] J. Ferber, O. Gutknecht, and F. Michel:. From agents to organizations: An organizational view of multi-agent systems. In Paolo Giorgini, Jörg P. Müller, and James Odell, editors, *IV International Workshop on Agent-Oriented Software Engineering (AOSE'03)*, volume 2935 of *LNCS*, pages 214–230. Springer–Verlag, 2003.

[7] N. Francez and I. Forman. *Interacting processes: A multiparty approach to coordinated distributed programming*. Addison–Wesley, 1996.

[8] Jochen Fromm. *The Emergence of Complexity*. Kassel university press, 2004.

[9] N. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, 2001.

[10] A. Karageorgos and N. Mehandjiev. A design complexity evaluation framework for agent-based system engineering methodologies. In A. Omicini, P. Petta, and J. Pitt, editors, *Fourth International Workshop Engineering Societies in the Agents World*, volume 3071 of *Lecture Notes in Computer Science*, pages 258–274. Springer, 2004.

[11] E. A. Kendall. Role modeling for agent system analysis, design, and implementation. *IEEE Concurrency*, 8(2):34–41, April/June 2000.

[12] H. Mintzberg. *The Structuring of Organizations*. Prentice-Hall, 1978.

[13] J. Odell. Agents and complex systems. *Journal of Object Technology*, 1(2):35–45, July-August 2002.

[14] Object Management Group (OMG). Unified modeling language: Superstructure. version 2.0. Final adopted specification ptc/03–08–02, OMG, August 2003. www.omg.org.

[15] J. Peña, R. Corchuelo, and J. L. Arjona. A top down approach for mas protocol descriptions. In *ACM Symposium on Applied Computing SAC'03*, pages 45–49, Melbourne, Florida, USA, 2003. ACM Press.

[16] J. Peña, R. Corchuelo, and M. Toro. Representing complex multi-agent organisations in uml. In *IX Jornadas de Ingeniería del Software y Bases de Datos JISBD'04*, pages 159–170, Málaga, Spain, 2004.

[17] J. Peña, R. Corchuelo, and J. L. Arjona. Towards Interaction Protocol Operations for Large Multi-agent Systems. In M. Hinchey, J. Rash, W. Truszkowski, C. Rouff, and D. Gordon-Spears, editors, *Proceedings of the Second International Workshop on Formal Approaches to Agent-Based Systems (FAABS 2002)*, volume 2699 of *LNAI*, pages 79–91, NASA-Goddard Space Flight Center, Greenbelt, MD, USA, 2002. Springer–Verlag.

[18] T. Reenskaug. Modeling systems in uml 2.0 - a proposal for a clarified collaboration.

[19] T. Reenskaug. A methodology for the design and description of complex, object- oriented systems. Technical report, Center for Industrial Research, Oslo, Norway, November 1.988.

[20] T. Reenskaug. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, 1996.

[21] D. Snowden and C. Kurtz. The new dynamics of strategy: Sense-making in a complex and complicated world. *IBM Systems Journal*, 42(3):35–45, 2003.

[22] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison–Wesley, second edition edition, 2002.

[23] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.