 UNIVERSITY OF CALGARY	Course Number: SENG 609.22	Course Name: Agent-based Software Engineering
	Session: Fall, 2003	Department: Electrical and Computer Engineering
		Document Type: Tutorial Report

Computational Complexity and Agent-based Software Engineering

I. Introduction:

With the abundance in the growth of computers, there is necessity to manage and reuse information or knowledge that is distributed across systems to obtain desired goals intelligently. The key advances in program design and development techniques over the past decades are procedural abstraction, abstract data types and object-oriented programming (OOP). Agent is a self-contained problem-solving system capable of autonomous, reactive, proactive and social system, is yet another software engineering tool or technique.

Software engineering (SENG) for agent systems is at an early stage of development and because of the above-mentioned concepts is gaining acceptance among both academic and industrial community. Agent system development is currently dominated by informal guidelines, heuristics and inspirations rather than formal principles and well-defined engineering techniques [Far02]. If agent-based systems or technology is to be a success, then its software engineering aspects should be considered seriously. Agent-based systems are to complement OOP and are not to replace it. Thus, the most important issues of agent-based software engineering are to understand the situations in which agent solutions are appropriate and a principled but informal development technique for agent systems development.

This report is outlined as follows: in section II, I explain some of the complexities encountered by adopting agent-based system principles. In section III, I outline the agent-based software lifecycle and compare it with the traditional SLC of specification, design, verification and implementation, to alleviate the complexities. I conclude this report by suggesting some techniques that were not referenced by agent communities and some of the techniques, which can be drawn from other fields to provide solutions for the complexities found in agent-based systems.

II. Complexities:

There is no proof that agent systems will improve software engineering productivity. But for certain classes of problems, it can significantly improve software development process. An understanding that agents are logically perfect reasoners since they have infinite resources available for reasoning is wrong. No real agent has these properties. Thus, selection of attributes for logical reasoning even though difficult is a must during the design and implementation of agents.

Agents are generally useful for industrial-strength software, which are complex in nature, typically characterized by large number of parts with many interactions. But some form of regularity can be found in such complex systems. It takes the form of hierarchy and will be composed of inter-related sub-systems with self-hierarchy. Such hierarchical systems evolve more quickly since they evolve from simple systems if there are stable intermediate forms, than non-hierarchical systems. It will be possible to trace interactions among sub-systems and interactions within sub-systems in such complex systems. Thus, such complexities can be handled by decomposition, abstraction and organization.

Agents should act in pursuit of their objectives while maintaining an ongoing interaction with environment i.e., they have to be proactive and reactive where such designs are generally difficult. A balance should be struck between the two since too much inclination towards the former may risk agents taking irrelevant or infeasible tasks. On the other hand, highly reactive behavior means that the agent will cater the short-term needs only, which is against the principles of agent-based system.

Agents are inherently unpredictable because of their autonomy since the patterns and effects of interaction are uncertain and decided during run-time. Hence the number, pattern and timing of interactions cannot be predicted in advance. Other reasons leading to uncertainty are acceptance or refusal to requests posted by this or other agents. Emergent behavior of agents due to flexible interactions between components leads to sophistication, which is generally complex to analyze [Woolridge99].

Understanding the limitations of agents during design is important. Intelligence of agents are limited by the state of the art in that domain and agents cannot be equated to AI techniques, with the concept of computers thinking like humans. A basic understanding is that agents are not universal solution and proper delineation is important. Agents are generally difficult to maintain and should be applied only if necessary. Agents are generally multi-threaded software applications and generally recognized as complex classes of computer systems to design and implement. So, implementation practices of concurrent and distributed systems cannot be ignored. For single threaded applications, agents are not generally recommended. Designing agent architecture from the scratch is not recommended and usage of COTS design should be considered. For example, while building an agent-based email system which notifies the user by phone if he gets email from a specific id by text to speech, then already existing tool for this purpose should be used. The overhead of managing inter-agent communication is high and the benefits of using an agent-based solution should not outweigh its benefits.

Moreover, there are no widely used software platforms for developing MAS which provides all the basic infrastructure for message handling, tracing and monitoring, run-time management and so on. This leads to considerable time spent on implementing the infrastructure by building libraries and software tools. Exhibiting control over the freeness with which agent's interact is important since it may lead to chaos.

Other reasons leading to complexity or uncertainty of agent implementation in complex, distributed systems are the decision of whether the information an agent has is right or wrong. The agent's sensor may be faulty, the information may be out of date or the agent may have deliberately or accidentally given false information. The information an agent has is not directly available to other agents and do not have access to data structures of other agents. Thus, designing such nesting of information is difficult and agents in

competitive environments must make decisions under uncertainty, predict environment's parameters, predict other agent's future moves, and successfully explain self and the other agent's actions. Building quality knowledge base for the amount of information gained by an agent over time is also difficult due to lack of information and/or noise affects quality of decisions. Thus, decision making under uncertainty is too complicated and computationally expensive to be implemented [Far02].

The environments are assumed to be history dependent and the next state of an environment is not determined by the action performed by the agent and the current state of the environment. The actions made earlier also play a part in determining the current state, which leads to non-determinism and uncertainty about actions performed even if an agent is knowledgeable about other agents and environments. The design methodology suggested by Woolridge [Woolrdige00] assumes that every system is guaranteed to end with runs and have finite length. This assumption is flawed in situations where implementing agents in real-time continuous process control is impossible, even though applicable for multi-threaded applications.

Metrics to measure the complexity of MAS under various conditions can be classified as subjective and objective metrics. Subjective metrics use the technique of function point (FP) that accounts for algorithmic complexity can be used. Objective metrics measure the internal complexity of agent system [Far02]. But run-time determination of how many FPs will be used by agents to achieve goals is impossible to calculate during design time or even during implementation.

Testing the quality of knowledge base gained by an agent over time is a difficult process since the information stored may be wrong from the beginning. The same applies when designing and testing agents since the repetition of interactions cannot be guaranteed in such dynamic environments. Real-time distributed process control systems also suffer from such problems but the system can be brought to some stable state easily, which is not possible in the case of a MAS with changing organizational structures.

III. Agent-based Software Life Cycle:

To date, there has been little work on viewing multi-agent systems (MAS) as a software engineering discipline. This can be solved by modifying the essential concepts of traditional SENG to suite agent systems lifecycle; focusing in particular on specification, design, implementation and verification process [Jennings][Woolridge97]. The role of SENG is to provide structures and techniques that make it easier to handle the complexity. The typical SLC starts with requirements elicitation or specification phase followed by design, implementation and verification. I discuss some of the methods employed for these phases and draw a SLC model for agent-based system in this section.

Specification: In this section, I analyze the requirements for an agent specification framework. It can be classified into beliefs that agents have (the information they have about their environment, which may be incomplete or incorrect), the goals that agents try to achieve, the actions that agents perform and the effects of these actions and the ongoing interaction that agents have with each other and their environment over time [Jennings]. Rao-Geoff's belief-desire-intention (BDI) model is the best-known model that gives a good account of interrelationships between the various attitudes that together comprise an

agent's internal state and is an extension of temporal logic. Pragmatically, the specification languages of agent-based systems are more complex than the comparatively simple temporal and modal languages that are common in computer science.

Design: Most of the design approaches focus on the problem of decision making for fixed time bounds and computational resource constraints. The design problem starts with the specification of the environment in which agent must inhabit together with specification of tasks is defined, which is generally difficult to determine. Moreover, it cannot be guaranteed that an agent will succeed in all environments. Thus, an agent design problem involves both task and environment properties. Woolridge [Woolridge00] has assumed that the agents have a repertoire of possible actions available to them to change the state of environment with the environment itself as a finite set of states. Then, the system is designed as runs where the state changes are effected by actions of the agents on the environment. It is generally very difficult to capture all the states of the environment and all the possible actions of MAS. Moreover, his design is restricted to runs of finite length.

For specifying the tasks of an agent, Woolridge takes the approach of associating utilities (benefit or value) to states, and the task of agent is to bring the states that maximize utility. The disadvantage of this approach for design is that only the local states can be decided and run-time or long-term view will suffer when assigning utilities to states during design time. This is overcome by assigning utility to runs than assigning to states but is complicated by defining such utility functions. Another approach suggested by Woolridge is to divide the tasks into achievement tasks or liveness properties and maintenance tasks or safety properties. The former is specified by a number of goal states an agent wishes to bring about and is the commonly studied form of task in AI. The latter describes the states that an agent should not reach or the states that should be avoided.

Implementation: Specification and design is not the end of agent software development; it should be implemented as a system that is correct with respect to the specifications. Three possibilities suggested by Woolridge are discussed in this section [Woolridge97] for achieving agent-based systems.

The first is to manually refine the specification into an executable form via some principled but informal refinement process. For reactive systems refinement is not straightforward since it is difficult to specify the pre- and post-conditions to determine which program structures are required to realise such specifications. Two such implementation methods are Z language by Luck and d'Inverno and BDI.

The second method of implementation is the direct implementation of the system without refinement or providing proofs that the system works. In this method, the environment makes something true and the agent responds by doing something to bring the environment under its control, based on game-theory. METATEM is a programming language for MAS, based on the concept of "on the basis of the past do the future".

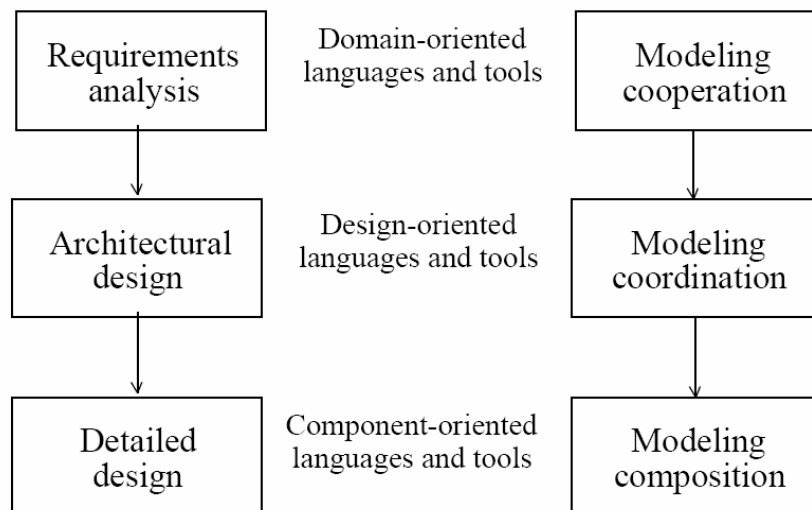
The third method is to compile the agent specifications into concrete model via automatic synthesis; the advantage being run-time efficiency. Direct execution of reasoning as in METATEM is computationally costly. Thus, compilation approach aims to reduce abstract symbolic specifications to simple computational model with reasoning done offline at compile-time. Automata theory, using finite state machines (FSM) is used to model the system given a specification. Timed automata machines are also possible.

Verification: Verification is to ensure that the design can guarantee to carry out the task successfully and the main question is how to decide whether an agent has succeeded. Again, it can be split into verification of achievement tasks and maintenance tasks given an environment. The execution histories or runs should be traceable to confirm this. In general, we can say that given an environment, agent and task specification, the answer should be ‘yes’ if every terminated run of agent in environment satisfies specification and ‘no’ otherwise. Woolridge uses the same run theory for verification of agents [Woolridge02].

Woolridge suggests two approaches to verification: axiomatic approach and semantic approach. Axiomatic approach uses systematic derivation of logic that represents the behaviour of program to prove problem to show that the implementation is correct. But reactive systems are infinite sequences and systematic derivation of the theory of program from text and are hard. The second is the semantic approach which is model checking based on the semantics of the specification language. Even though simple, this is essentially syntax checking for FSM given the fact that specification languages are not good and is hard to derive the beliefs, desires and intentions of the agents.

In all, the agent oriented software engineering process can be shown as cooperation model, coordination model and composition model shown in figure 1 [Ciancarini00].

Fig.1: Agent oriented software engineering process



Conclusion:

While the importance of agent-based software engineering is felt in academic and industrial setting owing to the increase in complexity of distributed systems for information sharing and reuse, agent-based systems should consider the solutions provided by other engineering or scientific fields to manage such complexities.

Taxonomic classification to characterize complexity in terms of number of interacting entities; number of states, parameters, and uncertain variables, the degree of uncertainty; degree and level of interaction and dependence; rates of evolution of the contributing technologies; level of maturity and history of the knowledge are essential to fully understand and design agent societies [Govindaraj98].

Other methods used for modeling such distributed complex systems with temporal constraints are Petri-nets approach designed by Petri [Petri], from Net-theory for solving problems in genetics. Petri-nets are used for the design of real-time distributed control systems and versions for considering temporal constraints are also available [Ling00].

Reference:

[Ciancarini00]: P. Ciancarini, "Components, agents, and architectures: a software engineering perspective",

[Far02]: B.H. Far, "Software agents: Quality, complexity and uncertain issues", IEEE conf. on CI, 2002, p- 1-10

[Govindaraj98]: T. Govindaraj, "Characterizing complexity in the design of human-integrated systems", IEEE, 1998, p- 985-989

[Jennings]: N.R. Jennings and M. Woolridge, "Agent-oriented software engineering", Queen Mary & Westfield College, U of London

[Ling00]: S. Ling *et al.*, "Time Petri nets for workflow modelling and analysis", IEEE Systems, Man, and Cybernetics, vol. 4, Oct. 2000, p 3039 –3044

[Petri]: http://www.informatik.uni-hamburg.de/TGI/pnbib/p/petri_c_a.html

[Woolrdige00]: M. Woolridge, "The computational complexity of Agent design problems", IEEE, 2000, p– 341-348

[Woolridge02]: M. Woolridge, "The computational complexity of Agent Verification", Springer-Verlag, 2002, p– 115-127

[Woolridge97]: M. Woolridge, "Agent-based software engineering", IEE Proc.- SENG, vol. 144, issue 1, 1997, p– 26-37

[Woolridge99]: M. Woolridge, "Software engineering with Agents: Pitfalls and pratfalls", IEEE Internet comp., 1999, p– 20-27