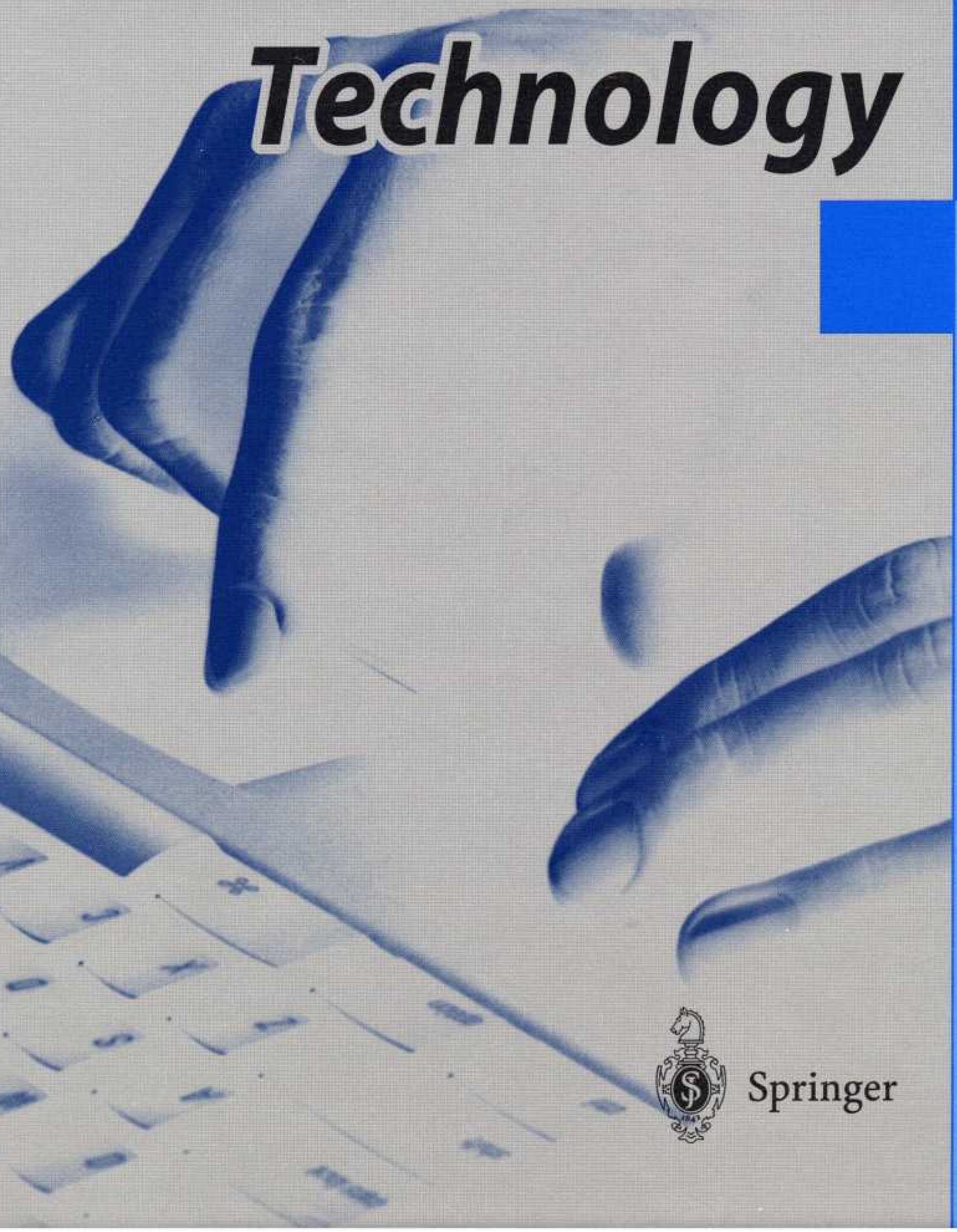
Q. Jin • J. Li • N. Zhang • J. Cheng
C. Yu • S. Noguchi (Eds.)

Enabling Society with Information Technology



Springer

Agent-SE: A Methodology for Agent Oriented Software Engineering

Behrouz H. Far

Faculty of Engineering, Saitama University,
255 Shimo-okubo, Saitama 338-0825, Japan

Summary. Agent-oriented approach to software engineering (Agent-SE) for multiagent software design is presented. It includes methods to generate organizational information for cooperative and coordinative agents. Agent-SE can be used to design and implement complex, heterogeneous, distributed and networked software systems using dynamic agent coalition structure.

1 Introduction

Software development is a very high risk task. Only about 1/6 of software projects are successful and almost all of the software projects' costs exceed initial estimation. In order to manage such a crisis, software engineering paradigms have been evolved from device oriented assembly languages to procedural and structured programming, to Object Oriented programming, to distributed objects and component-ware and to design patterns. Due to the increasing popularity of the Internet, heterogeneous, scalable and networked software systems are highly needed. However, neither of the software engineering paradigms could make software technology keep up with the current business needs.

Nowadays, an increasing number of software projects are being revised and restructured in terms of software agents. Software agents are considered as a new experimental embodiment of computer programs and are being advocated as a next generation model for engineering complex, heterogeneous, scalable, open, distributed and networked systems.

Agent system development is currently dominated by informal guidelines, heuristics and inspirations rather than formal principles and well-defined engineering techniques. Unfortunately, there has been comparatively little work on viewing agent system design and development as a software engineering paradigm that has the potentiality to enhance software developments in a wide range of applications. There is no standard way of incorporating agent-oriented viewpoint into design and development of agent-based software systems.

In this research we argue that the development of heterogeneous, robust and scalable software systems requires software agents that can complete their objectives while situated in a dynamic and uncertain environment, engage in interactions with other agents or humans and operate within flexible

organizational hierarchies. We also argue that agents can be used as a framework for bringing together the components of Artificial Intelligence (AI) and Software Engineering (SE) that are necessary to design and build intelligent artifacts.

The structure of this paper is as follows. In Section 2 we discuss how scalability and complexity can be handled in agent systems. In Section 3 the Agent-SE approach is introduced. In Section 4 a method to derive organizational knowledge is presented. Finally, a conclusion is given in Section 5.

2 Software System Complexity

Business software has a large number of parts that have many interactions (i.e., complexity). The role of software engineering is to provide techniques that make it easier to handle such complexity [6].

2.1 Complexity in SE

Complexity in software systems is structural in nature. As a software system evolves new functions, program modules are added to it and its structure deteriorates to the extent that major effort is needed to maintain its consistency and conformity with the specification. Therefore, hierarchical design is a major way of handling complexity in software engineering. A complex software system is usually composed of many interrelated subsystems, each of which is in turn hierarchic in structure. The relationships among the components are supposed to be dynamic.

Two kinds of relationships can be devised: interactions among subsystems and intra-actions within subsystems. Interactions are between an artifact (or a component) and its outer environment. Intra-actions are the characteristics of the artifact's (or a component's) inner environment.

Contemporary software engineering techniques can manage only the intra-actions using decomposition and abstraction techniques. Decomposition is dividing a large problem into smaller, more manageable units each of which can then be dealt with in relative isolation. Abstraction is to define a simplified model of the system that emphasizes some of the details or properties, while intentionally neglecting the others. Attention can be focused on some aspects of the problem, at the expense of the other less relevant details. All of the contemporary software engineering paradigm, such as: object-orientation, component-ware, design patterns and software architectures provide techniques for decomposition and abstraction.

2.2 Complexity in AI

Complexity in AI is handled via using ontologies and applying synthesis techniques. Ontology in AI refers to a set of concepts or terms that can be used

to describe some area of knowledge or build a representation of it. Interest in ontologies has grown due to interests in reusing or sharing knowledge across systems. Developing reusable ontologies that facilitates sharing and reuse is a goal of ontology research [2]. We think that ontologies can play a significant role in identification and design of interactions among software agents (see Section 3). Synthesis works exactly opposite to decomposition. In synthesis, one first defines a subclass of problem to be solved and builds a simplified model or prototype system that will be later incrementally updated to account for additional properties. We think that synthesis is a practical technique for building coalition of software agents.

3 The Agent-SE Approach

In this section we propose a method for multiagent system design, called Agent-SE, based on the abstraction and decomposition (Section 2.1), ontology and synthesis (Section 2.2) and organizational properties. The Agent-SE design steps are as follows:

1. Decompose the problem based on function/ input/ output into an organization of agents.
2. Design the task ontology of the problem.
3. Build an abstraction model and add interactions and signal level organizational relationships using the task ontology.
4. Design each agent and its internal intra-actions using conventional SE techniques (preferably, object-oriented design with UML, etc.) and a predefined agent model, if necessary.
5. Based on the domain ontology, design each agent's knowledge-base using Symbol Structure. (See Section 4).
6. Derive and record symbol level organizational properties based on interactions of pairs of cooperative or coordinative agents. (See Section 4).

These steps are explained in detail in the following sections. Agent-SE offers:

- An effective way of decomposition (partitioning the problem space) and synthesis.
- A means of introducing abstraction to the model.
- An appropriate way of modeling and viewing organizational relationships of complex systems.

Some novel points are:

- Decomposition is based on the function and input/ output rather than conventional data/object.
- Participating agents are described by specifying their input/output and/or functions (interfaces) or their inner environment (classes).

- Organizational properties are derived dynamically.

Organizational formation, maintenance and updating are typical of the dynamic nature of groupings in complex systems. Agent-based systems require computational mechanisms for dynamic formation, maintenance and disbanding of organizations. One such mechanism is presented in Section 4.

4 Organization

Organization is a goal directed coalition of software agents in which the agents are engaged in one or more tasks. Control, knowledge and capabilities are distributed among the agents.

Organizations, of various forms, physical, cognitive, temporal and institutional have been studied in management and computer sciences. The game theoretic approach to study organization focuses on modeling and suggesting computational algorithms for certain aspects of the coalition, such as social welfare, individual rationality, voting consensus, etc. The computational approach focuses on identifying general principles of organization and their exceptions.

The already proposed organizational models for multiagent systems have certain drawbacks. First, they cannot explain the organizational knowledge in terms of its comprising agents without reference to any other intermediary concepts. Second, they cannot provide frameworks for comparing and evaluating different organizations. Third, the organizational knowledge base cannot be updated dynamically, accounting for different configuration of the participant agents. Finally, they cannot explain the need for services of a certain agent in an organization. All of these factors are necessary in organization design and are addressed in our research.

4.1 Assumptions

Intelligence of Pair (IoP) The already proposed theories and formalisms have implicitly assumed that Organizational Intelligence (OI) exists and implemented using a meta-agent (e.g., directory and ontology service agents) (such as [1]). However there are certain difficulties in both logical formulation and actual implementation of such theories. This is mainly due to ignoring the dynamic interactions among the agents when devising the components of OI.

A point in our research is that in a purposeful (i.e., not random) organization, OI is a property of interaction among agents and can only be ascribed to at least a pair of agents. We call this Intelligence of Pair (IoP) assumption.

History of Patterns (HoP) In biological coalitions, participants may have a kind of role or function (during interaction with the other participants), if

they show some persistence in their profile of actions over time. The same could be devised for artificial coalitions. As a matter of fact, it is not difficult to find organizations that display non-random persistent and repeated patterns of actions [1].

Agents act and perform in a physical world. Their past experiences can be recorded and explained in terms of their histories, that is, their profile of actions and states that they go through. Intuitively, histories can display certain patterns. A basic feature of state representation is that it assigns a certain characteristic to its reference agent. Therefore it is possible to define OI patterns with reference to agents' history.

Another point is that OI patterns emerge from discovering a persisted state or an ordered pattern in the agent's profile of actions. We call this History of Patterns (HoP) assumption. IoP and HoP assumptions account for dynamic interactions and a computation method based on this assumption is proposed below.

4.2 Modeling

Symbol structure (SS) is used to model individual agent's knowledge structure. SS is a finite connected multi-layer bipartite graph. There are two kinds of nodes in each layer of SS: concepts (c) and relations (r). One source of difficulty when processing concepts, is distinguishing a concept at various levels of abstraction, as well as differentiating between generic concepts and their instances. Function *type* is defined to ease such differentiation. The function type maps concepts and relations onto a set T. The elements of T are called type labels. Type hierarchy provides a means of evaluating a concept at various levels. The type hierarchy is a partial ordering defined over the set of type labels, T.

Flexibility, extendibility and interoperability are three main advantages of knowledge representation and reasoning with SS.

4.3 Reasoning rules

Join rule: Join rule merges identical concepts. If a concept c in symbol structure u is identical to a concept d in symbol structure v, then let w be the symbol structure obtained by deleting d and linking to c all arcs of relations that had been linked to d.

Simplification rule: Redundant relations of the same type linked to the same concept in the same order can be reduced by deletion all but one. If the relations r and s in the symbol structure u are duplicates, then one of them may be deleted from u together with all its arcs.

Generalization/Specialization rule: For two arbitrary levels u and v of any SS, if u is identical to v except that some type labels of the nodes of v are restricted to subtypes of the same nodes in u, then u is called a specialization of v, and v is called a generalization of u.

4.4 Interaction among agents

Now we have a framework for representing and reasoning with the knowledge on an individual agent basis. Knowledge sharing by moving from one agent to another and on an organizational basis requires defining the basic agent interactions, i.e., cooperation, coordination and competition.

Cooperation: Cooperation is revealing an agent's goal and the knowledge behind it, i.e., its symbol structure to the other party. In cooperation both agents have a common goals.

Coordination: Coordination is revealing an agent's goals and the knowledge behind it, i.e., its symbol structure to the other party. In coordination, agents have separate goals.

Loose Competition: Loose competition is revealing only an agent's goals but encapsulating the knowledge behind it to the other party.

Strict Competition: Strict competition is neither revealing an agent's goals nor the knowledge behind it to the other party. Therefore, knowledge sharing is equivalent to merging two or more symbol structures using join, simplification, generalization and specialization rules.

Conventionally, it is believed that for a pair of agents to interact, each should maintain a model of the other agent, as well as a probabilistic model of future interactions [5]. This is totally unnecessary when using SS representation in cooperation and coordination cases.

4.5 Computational OI

Here we propose a method for generating organizational concepts based on the IoP and HoP assumptions and definitions of interaction. In case of *cooperation* and *coordination* the agent's private knowledge is exposed to the other party. In this method, first, a pair of agents are selected and by using join, simplification, generalization and specialization rules their SS are merged. There are finite sets of *actions* and *states* associated with each agent. Given a set of all possible actions, A, an agent's action is a subset $B \subseteq A$. Each agent's state transition is represented by a non-deterministic finite state automaton (NFSA).

Both *actions* and *states* take concepts and relations as their attributes. For each agent, the NFSA state transition model has an initial state *idle* and final state *goal*. A sequence of actions that convert the *idle* state to *goal* is a plan of actions towards a goal. Theoretically, all such sequences form a *regular language (RL)* whose elements are generated by a *regular grammar (RG)* equivalent to the NFSA. Furthermore, there may be various modes of operation (i.e., authority mode, subordinate mode, etc.) each modeled by a different NFSA. The proper mode may be selected by examining the select mode (See Fig. 1).

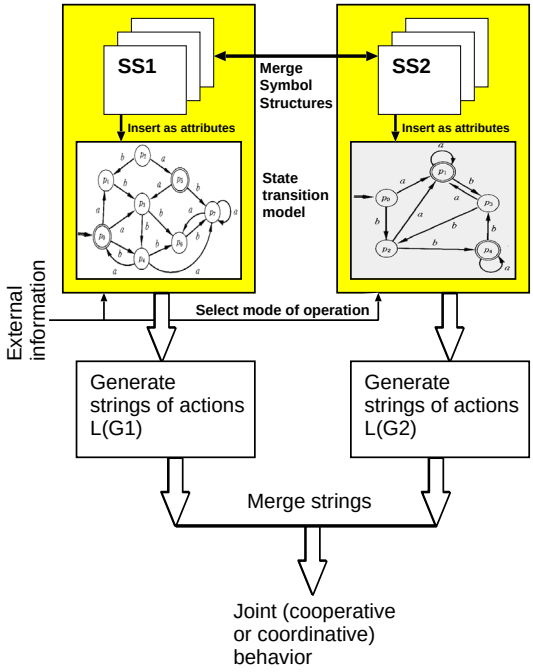


Fig. 1. Deriving organizational knowledge

As both actions and states take concepts and relations of SS as their attributes, in the case of cooperation and coordination, a joint sequence of actions can be generated by matching the actions whose attributes are concepts or relations belonging to the same type hierarchy. Such sequences form the joint plans of actions handled partially by either of the agents pair. The proposed algorithm is as follows:

1. Select an agent pair, Agent (G_1) and (G_2).
2. Merge their SS using join, simplification, generalization and specialization rules.
3. Select mode of operation based on external information.
4. Select the NFSA model and generate sequences of actions for the selected mode.
5. Compare two such sequences $\omega_i \in \mathcal{L}_{G_1}$ and $\omega_j \in \mathcal{L}_{G_2}$ where $\mathcal{L}_{G_1}$ and $\mathcal{L}_{G_2}$ are regular languages of agent (G_1) and (G_2), respectively.
6. For a common action $a \in \omega_i$ and $a \in \omega_j$ check the type of attributes of a. If the attributes belong to the same type hierarchy, merge the sequences from that point on after adjusting the types.
7. Record such joint sequences and check for possible repetition and/or persistence patterns in the future course of actions.

8. When repetition and/or persistence becomes present, add such strings to the *organizational knowledge base*.

Examples of our *electronic commerce* project [3] are presented below. In our *electronic commerce* project 7 types of agents, *customer*, *dealer*, *manufacturer*, *delivery*, *banker*, *search* and *catalog* agents work together and/or compete to do business on the Web [3]. By default, the *dealer* agent only knows about the goods to be sold, the *delivery* agent knows about transporting goods, *banker* agent has information on customers and their credit and/or cash accounts and *customer* agent is a personal assistant agent for a human user and has information on the user's preferences, etc.

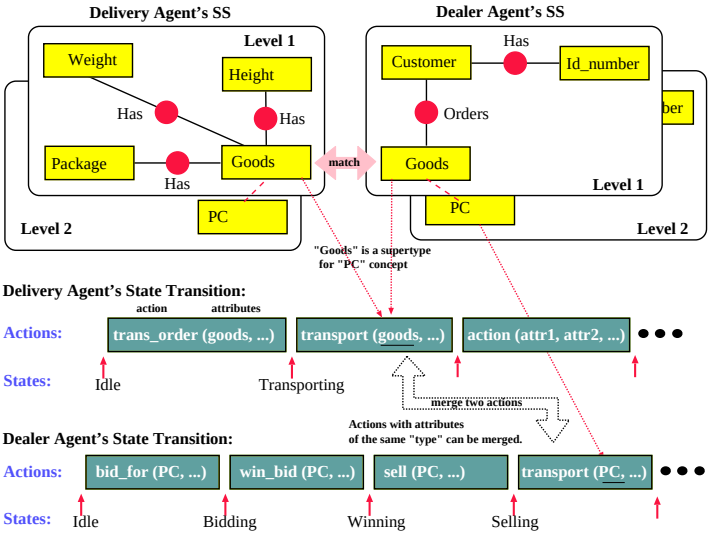


Fig. 2. A portion of SS for dealer agent

4.6 Example: cooperation

Let's consider a case of a *dealer* agent and a *delivery* agent cooperating to sell and deliver an article of commerce, such as a PC to a human user. A portion of SS for *dealer* and a *delivery* agents is shown in the upper portion of Fig. 2. It is visible that the concept `Goods` is a super-type for `PC` for both agents. However, based on the agents' roles, the concepts may have different data associated with them. For example, `PC` in *dealer* agent's SS may be associated with CPU, memory, etc. However, for *delivery* agent the same concept may have weight and size as its attributes.

The lower part of Fig. 2 depicts an example of the sequence of actions for the *dealer* and *delivery* agents. It is shown that the action `transport`(*Goods*,...) for the *delivery* and `transport`(*PC*,...) for the *dealer* agents have attributes belonging to the same type hierarchy. Therefore, the strings can be merged by adjusting the type and changing `transport`(*Goods*,...) to `transport`(*PC*,...) for the *delivery* agent and let the plan be executed by assigning this action to the *delivery* agent.

4.7 Example: coordination

Let's consider a case of a *dealer* agent and a *banker* agent coordinating their operations in order to exchange information related to a particular human user. Apparently the *dealer* and *banker* agents have separate goals but they coordinate their activities to help each other.

Let's assume that the *dealer* agent already receives a purchase order from a human user through his/her *customer* agent. A portion of the *dealer* agent's SS is shown in the upper right part of Fig. 2.

The *dealer* agent interacts with the *banker* agent by sharing its SS with that of the *banker* agent. By merging the two SS, the `customer` concept is common between the two SS and in this way the *dealer* agent can verify that the `customer` has an `Account` and from the *banker* agent's SS verify that there are either `Cash` or `Credit` accounts available. Also it can verify that the user may also use a `Cash Card`. Therefore, the *dealer* agent can contact the user to get data for `Cash` or `Credit` the accounts. It can further verify genuineness of the data supplied by the user by consulting the *banker* agent again. In this simple example, knowledge sharing is used to enable the *dealer* agent to successfully proceed with the selling task in spite of possessing only a limited amount of knowledge about the user without implementing a redundant customer database within the *dealer* agent.

5 Conclusions

In this paper, we proposed a method for multiagent system design by blending AI and SE techniques. We defined agents' interaction problems as cooperation, coordination and competition and devised a method to extract organizational knowledge during cooperation and coordination. These methods are quite useful for designing agent coalitions that are shaped dynamically and defining multiple roles for individual agents when participating in a number of different coalitions.

We have argued that multiagent system design can be achieved at the expense of additional design steps including, design of domain ontology and agents' symbol structure. Actually, designing domain ontology is not a totally new task in software engineering practice. It has been carried out informally in almost all of the contemporary software engineering techniques. For instance,

data definition diagrams incorporated with flowcharts and PAD diagrams in structural design and actor definitions in use-cases in object-oriented design are kinds of treatments of this step. We insist on the significance of formalization of this step using task level ontology and the need for tools to support this.

Applications using the framework and techniques described in this paper, such as a multi-agent system for electronic commerce [3,4] have already been developed.

References

1. K.M. Carley and L. Gasser, "Computational Organization Theory," in Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, ed. G. Weiss, G., pp. 299-330, MIT Press, 1999.
2. B. Chandrasekaran, et al., "What Are Ontologies, and Why Do We Need Them?" IEEE Intelligent Systems and Their Applications, vol. 14, no. 1, pp. 20-26, 1999.
3. B.H. Far, et al., "An Integrated Reasoning and Learning Environment for WWW Based Software Agents for Electronic Commerce," IEICE Trans. Inf. and Syst., vol. E81-D, no. 12, pp. 1374-1386, 1998.
4. B.H. Far, et al., "Formalization of Organizational Intelligence for Multiagent System Design," IEICE Trans. Inf. and Syst., vol. E83-D, no. 4, pp. 599-607, 2000.
5. M.N. Huhns and L.A. Stephens, "Multiagent Systems and Societies of Agents," in Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, ed. G. Weiss, pp. 79-120, MIT Press, 1999.
6. N.R. Jennings, "On agent-based software engineering," Artificial Intelligence, vol. 117, pp. 277-296, 2000.