TRANSACTION-BASED COORDINATION OF SOFTWARE AGENTS

E. PITOURA

10-98

Preprint no. 10-98/1998

Department of Computer Science
University of Ioannina
451 10 Ioannina, Greece

Transaction-Based Coordination of Software Agents*

Evaggelia Pitoura
Department of Computer Science
University of Ioannina
GR 45110 Ioannina, Greece
Phone : + 30 (651) 97 311
Fax : + 30 (651) 48 131
Email: pitoura@cs.uoi.gr

Abstract

Cooperative software agents provide a novel framework for building distributed applications. Central to the model is the support for sophisticated coordination and close cooperation among different agents working together towards accomplishing a specified task. In this paper, we focus on expressing and enforcing correctness properties of the coordination and interaction among agents. We build on transaction concepts from database systems to formalize the proposed correctness properties and introduce methods for enforcing them in the context of mobile object systems.

Keywords: mobile objects, transaction models, concurrency control.

1 Introduction

Mobile agents are programs which may be dispatched from a client computer and transported to a remote server computer for execution. With the rapid development of network-centered applications and web-based technologies, this new model of distributed computing has attracted much attention. The mobile agent model provides an efficient, asynchronous method for attaining information or services in rapidly evolving networks: mobile agents are launched into the unstructured network and roam around to accomplish their task. Agents simplify the

*University of Ioannina, Computer Science Department, Technical Report No: 98-10

necessary distribution of computation and overcome the communication barrier by minimizing the interchanged messages. In addition, mobile agents support intermittent connectivity, slow networks, and light-weight devices which is the case in wireless computing since wireless networks are less reliable and offer less bandwidth than their wireline counterparts and portable machines are resource poor when compared to stationary servers. In wireless computing, mobile agents can be used to offload functionality from wireless components to the fixed network [PS98].

Most current research on agent-based models [VT97, IEE97, CAC94] focuses on aspects related to intelligence and security, both critical for the realization and success of mobile agent systems. In this paper, we address another issue, that of consistency. Agents model long-lived computations, own local resources, and access data at remote systems. There is a need to ensure that the concurrent execution of multiple agents does not violate the consistency of the data accessed. Besides their own data, agents access two types of data. First, they access data owned by other cooperative mobile agents, that is data that belong to the context of other concurrently executing agents. This form of interference among agents must be controlled to ensure the consistency of each agent's context in the presence of concurrent accesses. Second, agents query and modify data that belong to remote systems. Remote systems may include persistent data stores such as databases.

Ensuring consistency in the context of the agent model is hard since there is no central point for the coordination of agents. In addition, the behavior of agents is dynamic, and there may be no prior knowledge of their potential interaction. In this paper, we consider agents in the context of object systems. An agent is an active object with its own methods and data.

Our approach is based on associating with each agent an *agent manager* to coordinate its execution. An agent manager provides a focal point for the coordination of each mobile agent. Each agent manager is itself an agent that may be mobile and distributed at different sites. The special methods that the agent manager supports form a small and uniform for all

managers set of primitives. Thus, this set can be implemented as part of an agent library. Agent managers are the analog to transaction managers in traditional database concurrency control. They cooperate with each other to enforce the serialization of the executions of their associated agents. The proposed technique provides fine-grained control over agent execution at the level of each agent's method execution.

The remainder of this paper is organized as follows. In Section 2, the structure of the execution of an agent is defined as well as constructs to provide cooperation between agents. In Section 3, a transaction model for agents is presented, and in Section 4, protocols for enforcing it are advanced. Related work is briefly summarized in Section 5. Finally, we offer our conclusions in Section 6.

2 Concurrency and Synchronization

Mobile agents are processes dispatched from a source computer to accomplish a specified task [CGH+95, Whi96]. Each *mobile agent* is a computation with its local data and execution state. After its submission, the mobile agent proceeds autonomously and independently of the sending client. When the agent reaches its destination processing unit, it is delivered to an agent execution environment. Then, if the agent possesses necessary authentication credentials, its executable parts are started. To accomplish its task, a mobile agent can transport itself to another computer, spawn new agents, and interact with other agents. Upon completion, the mobile agent delivers the results to the sending client or to another server.

In the following section, we abstract the constituting parts of an agent, and formalize intra-agent and inter-agent synchronization.

2.1 The Model

An *agent* is an active object encapsulating the state, behavior and location of a computation. The state of an agent is the set of values of a set of local to the computation variables; its

behavior is the set of methods it provides; and its location is the context of its execution. Computation starts by invoking an appropriate agent. For a processing unit to execute an agent, it must provide an agent execution environment.

The agent may be executed as an operating system process, an operating system thread or it may be managed by a thread package within the agent execution environment. The methods that build the agent program may correspond to local functions to the agent, functions of other agents, or functions of remote systems.

To develop our model, we distinguish the methods of an agent as simple and composite. Composite methods consist of a number of other simple or composite methods. A simple method is either a primitive or a basic method. *Primitive* is a method that only accesses the state of the agent, that is a method that manipulates only local variables. *Basic* is a method that accesses non-local to the agent resources, that is resources that belong to another agent or remote system. Such methods are either methods that access resources of another mobile agent or resources of the remote system where the agent is transmitted. Each agent has a top-most method, called *Compute*, that encodes the intended computation.

Agents are instances of a particular system-defined class called $Agent\_Class$, thus all agents have similar behavior and structure. Upon activation of an agent, the agent creates an Agent Manager (AM) to coordinate its execution. AMs are special agents, instances of a system class called AM_Class. Specific local variables of an AM represent the execution state of each method of the corresponding agent. An additional local variable records the history of the execution of the agent, in particular the order of method invocations.

2.2 Intra-agent Concurrency and Synchronization

Concurrency inside an agent is achieved through asynchronous invocations of more than one of its methods. Synchronization mechanisms are necessary to control the interaction among the execution of the agent's methods. Intra-agent synchronization is based on defining a set

of dependencies on the ordering of the execution of methods [RS95a]. Such dependencies are called structural and are expressed in terms of the controllable states of the methods of the agent.

The states of a method include the submission, execution and completion of the method. When a method is invoked, it enters the ready state and moves to the submitted state when it is actually submitted for execution. In terms of basic methods, the agent manager of an agent may delay the submission of a method to another agent or system to ensure correctness. When its execution starts, the method enters the running state and remains in this state until its completion. Upon completion, the method may be either accepted or aborted. If accepted, the results of the method become permanent and the method enters the committed state. The committed state may either correspond to failure or success depending on whether the expected result was accomplished or not. If aborted, the results of the method are not recorded, and the method has no effect on data.

A state of a method is controllable, if the agent that submitted the method or its agent manager can cause a transition of the method into this state. Although, the agent that invokes a primitive method and its manager can control all states of a method, they can control only the submission of a basic method. The actual execution time of a basic method is under the control of the corresponding remote mobile agent or system that receives the agent. The same holds for the completion of a basic method. Some remote systems may provide a prepare-to-commit state that indicates that a method has completed execution and its results are about to become permanent. If this feature is available, then the agent or its manager can also decide on the completion of the method. Figure 1 summarizes the controllable states of a basic method. The following definition formalizes the concept of a structural dependency among states:

Definition 1 (structural dependency) *A structural dependency SD is a triple $(C, \mathcal{M}, \mathcal{S})$, where C is a specification, $\mathcal{M}$ is a set of methods M, and $\mathcal{S}$ is a set of controllable states of the methods $M \in \mathcal{M}$.*

Figure 1: Controllable states for basic methods, when: (a) there is no prepare-to-commit state, (b) there is a prepare-to-commit state

There are three types of structural dependency based on the form of C: order, existence, and real-time dependencies in accordance with the primitives defined in [Kle91]. In an *order structural dependency* (C, $\mathcal{M}$, $\mathcal{S}$), C has one of the following forms: M_i can enter state s_i only after M_j has entered state s_j, or M_i cannot enter state s_i after M_j has entered state s_j, where M_i, $M_j \in \mathcal{M}$ and s_i, $s_j \in \mathcal{S}$. Ordering structural dependencies can be used to express data flow dependencies, for instance that M_i reads data produced by M_j. Ordering dependencies can be used to enforce a serial execution of a number of methods, by forcing a method to begin only after the commitment of the previous one. In an *existence structural dependency* (C, $\mathcal{M}$, $\mathcal{S}$), C has the following form: if M_i enters state s_i, then M_j must enter state s_j, where M_i, $M_j \in \mathcal{M}$ and s_i, $s_j \in \mathcal{S}$. Special cases of existence structural dependencies include critical, contingency, and compensation methods. *Critical methods* are methods that, when aborted (or failed semantically) cause the entire agent to abort (or fail semantically). *Contingency methods* are methods that are executed as alternatives when an agent fails semantically. *Compensation methods* are methods that are executed to semantically undo the effect of a committed method

when some other method aborts. Finally, in a *real-time structural dependency* $(C, \mathcal{M}, \mathcal{S})$, C specifies a requirement for the real time submission or completion of the methods in $\mathcal{M}$.

The state of a method is stored as a local variable of the associated AM. The value of this variable is modified when a state transition occurs. To implement structural dependencies, the agent manager handles explicitly the controllable state transitions. This is achieved through the *form_dependency* method. The form_dependency is a method of the AMs. It takes as arguments the type of the dependency and the names of the associated methods and states. Upon receipt of a form_dependency method, the AM becomes responsible for the enforcement of the dependency. The dependency may be formed during the execution of an agent based on the result of a previous action. Dependencies may be also specified statically as part of the definition of an agent.

Defining flow of control and data using dependencies among states of methods provides fine-grained control over the execution of an agent. It also allows for a large degree of parallelism in the agent's execution, which is very important since different parts (methods) of the agent may be executed at different network sites.

2.3 Inter-agent Communication and Synchronization

Concurrency at the inter-agent level is achieved by allowing more than one agent enactment. Agents interact with each other to accomplish a common goal. Moreover, the execution of an agent may interfere with the execution of another agent indirectly when they both access the same data concurrently. There is a need to define and control both the desirable and indirect interactions between agents that are executed concurrently.

Inter-agent synchronization and communication can be achieved either through message passing or through shared memory. Message passing provides explicit control on the visibility of an agent's data, since an agent explicitly sends information about its local data to other agents or accepts similar information about other agent's data from them. Both asynchronous

and synchronous messages from an agent to another may be provided.

```
<send> Object <to> Agent-name

<receive> Object <from> Agent-name
```

Interaction through shared memory relies on the observation of partial changes in an agent's local data caused by other agents. This interaction can be controlled by defining specific points in the execution of an agent where other agents are allowed to observe its partial results through modifications in the agent's local data. This may be accomplished by explicitly defining break [FO89] or permit [BDG+94] points within the execution of an agent where other agents are allowed to interleave.

Definition 2 (breakpoint) *A breakpoint B of an agent T is a triple $(B_s, B_e, \{(T_i, M_j)\})$ where $\{(T_i, M_j)\}$ is a set of pairs of agents (T_i) and methods (M_j) of these agents, and $B_s = (M_s, s_c)$, $B_e = (M_e, s_e)$ are pairs of methods along with corresponding controllable states of the agent T which allow members of $\{(T_i, M_j)\}$ to be executed between B_s and B_e of T.*

Another form of inter-agent communication is *delegation*, where an agent delegates responsibility of the execution of a method to another agent.

Definition 3 (delegation) *A delegation of an agent T is a pair (M_i, T_j) that denotes that the method M_i invoked by T will be executed as part of agent T_j.*

To implement breakpoints and delegation, two special methods, called *breakpoint* and *delegation*, are used. A *breakpoint* method is sent to the Agent Managers of the agents that are involved in the breakpoint. The *delegation* method is sent to the Agent Manager of the agent to which the execution of the method is delegated.

Interaction among agents can also be achieved by defining a common to all agents data storage. Agents can then communicate by importing and exporting data from this common storage. The techniques in this paper are applicable to this scenario as well by considering the common storage as the local data of a special agent.

3 The Agent Transaction Model

Attributing to agents the properties of traditional transactions is too restrictive. In particular, the execution of each agent is not isolated from the execution of other agents, since agents can see the intermediate results of execution of other agents by accessing their local data. Furthermore, the execution of an agent is not necessarily atomic, since while a method of the agent may be aborted, some other method may be accepted. In this section, we define what is a correct execution of an agent as well as a correct execution of a number of concurrently executing agents.

3.1 Well-Structured Agent Execution

Two methods of an agent conflict if they do not commute, that is if the order of their execution affects the final state of data. A commutativity relation is defined for each pair of methods. Two methods conflict if they do not commute. For example, two methods conflict if they access the same data item and one of them updates it. We assume that a commutativity relation is defined for both simple and composite methods. Alternatively, commutativity can be defined only for simple (primitive and basic) methods and the commutativity of composite methods can be inferred by the commutativity of their constituting simple methods. In a closed-nested transaction model, such as that in [HH91], conflicts among primitive or basic methods result in conflicts among the composite methods from which they are invoked. In open-nested transactions [MRW+93], there is no such implication. Although, we assume for clarity of presentation, such an open-nested transaction model, most of the following definitions and protocols translate easily to the closed-nested situation by for instance using such techniques as hierarchical timestamps [HH91].

The computation of an agent is extended to include the methods that are delegated to it. In particular, the execution of an agent is modeled as a sequence of its breakpoints and of state transitions of the methods that it invokes or are delegated to it. Thus,

Definition 4 (agent execution) *An* agent execution *is a pair* $(\Sigma, <)$ *where* Σ *is a set of events and* $<$ *is a partial order. An event in an agent execution is either a breakpoint* B *in the Compute of the agent or a pair* (M, S) *where* M *is a non-delegated method invoked by the Compute method of the agent or a method delegated to it and* S *is a state of* M*. The partial order* $<$ *is such that for all non commutable methods* M_i *and* M_j *either* $(M_i, E) < (M_j, E)$ *or* $(M_j, E) < (M_i, E)$*, where* E *stands for the running state of the method.*

Thus, the execution of an agent is a partially-ordered sequence of method executions and breakpoint events. This partial order must respect the structural dependencies among the methods of an agent. This is expressed in the following definition,

Definition 5 (well-structured agent execution) *An agent execution is* well-structured *in terms of a set* D *of structural dependencies if the partial order* $<$ *on its events does not violate any of the dependencies in* D*.*

3.2 Schedule Correctness

A schedule is an interleaved execution of methods and breakpoints of a set of agents. Formally,

Definition 6 (schedule) *A* schedule *of a set* $\{T_1, T_2, ..., T_n\}$ *of agents executions* $T_i = (\Sigma_i, <_i)$ *is a pair* $(\Sigma, <_h)$ *where* $\Sigma = \bigcup \Sigma_i$ *and* $<_h$ *is a partial order such that: (1) if, for any events* s_k *and* $s_l \in \Sigma_i$*,* $s_k <_i s_l$ *then* $s_k <_h s_l$*, and (2) for all non commutable methods* M_i *and* M_j *of two different agent executions either* $(M_i, E) <_h (M_j, E)$ *or* $(M_j, E) <_h (M_i, E)$*, where* E *stands for the running state.*

The first condition states that, the interleaved execution of a set of agents preserves the execution order of each of the agents. The second condition imposes a relative order between the non-commutable methods of two different agent executions.

Each agent or remote system observes the execution of agents by changes in the state of its local data. The projection of a schedule S on the local data of agent T_i is the schedule that

results if we exclude from S all but the primitive methods on data of T_i and the breakpoints, if any, that immediately precede each of them. Similarly, the projection of a schedule S on the local data of a remote system DB_i is the schedule that results if we exclude from S all but the basic methods on data of DB_i and the breakpoints, if any, that immediately precede each of them.

To enforce that the concurrent execution of transactions (agents) does not violate the consistency of data, the most common approach is to ensure that the corresponding schedule is conflict-serializable, that is conflict equivalent to a serial schedule [BHG87]. Two schedules are conflict equivalent, if they consist of the same events and order conflicting operations in the same manner. This is based on the assumption that each transaction (agent) maintains data consistency if executed alone. In the case of breakpoints, serializability is extended to relative serializability [FO89].

A *step* of an agent execution T is a subsequence of T that includes exactly the events between two consecutive breakpoints in T. A method interleaves with a step if it executed before an event E of the step and after another event E' of the step. We use a modified model of relatively serializability [FO89] that allows for efficient, i.e., polynomial, serializability testing [ABAK94]. We say that, in a schedule S, a method M_2 *directly depends on* a method M_1 if $(M_1, E) <_h (M_2, E)$, where E stands for the running state. The *depends on* relation is defined as the transitive closure of the directly depends on relation.

A schedule is *relatively serial* if, for all pairs of agents, T_i and T_j, if a method M of T_i is interleaved with a step of T_j starting with breakpoint $(B_s, B_e, \{(T_k, M_l)\})$ then at least one of the following is true: (a) M does not depend on any method pair of an event of the step and no method of an event of the step depends on M, or (b) $(T_i, M) \in \{(T_k, M_l)\}$.

Definition 7 (correct schedule) *A schedule is* correct *if (1) the execution of all its agents are well-structured, and (2) it is conflict equivalent to a relatively serial schedule.*

As the following theorem shows, to ensure that a schedule is conflict equivalent to some serial schedule, it suffices to ensure that its projections are conflict serializable with consistent orders:

Theorem 1 *If each agent and remote system projection of a schedule S is conflict equivalent to a relatively serial schedule, then, if there is an order $<_o$ consistent with the serialization orders assumed by each projection, then S is conflict equivalent to a relatively serial schedule.*

Proof. (Sketch) The schedule S_o having the same set of agents as S and with order $<_o$ orders conflicting operations the same way with schedule S, so it is conflict-equivalent to it. It is also relatively serial. To prove that, suppose it is not, then one of its projections is not relatively serial, a contradiction. □

4 Managing Agents at Run-Time

Ensuring the correctness of the interleaved execution of agents is the responsibility of different units, in particular of (Figure 2):

- agent managers (AM) of all agents involved

  The AMs are created upon the activation of each agent to ensure its structural correctness. They decompose the top-most method (*Compute*) of their associated agents into basic and primitive methods and take care of submitting the basic methods to the appropriate AMs or $DGTMs$ at remote sites.

- distributed transaction managers (DGTMs) located on top of each remote system

  Each DGTM coordinates the submission of agents to its system. Each DGTM receives methods from the various AMs, schedules them to control concurrency and inter-agent synchronization and submits them to the corresponding LTMs.

- pre-existing local transaction managers (LTMs) at each remote system

  An LTM at a remote system controls the execution of the basic methods submitted to

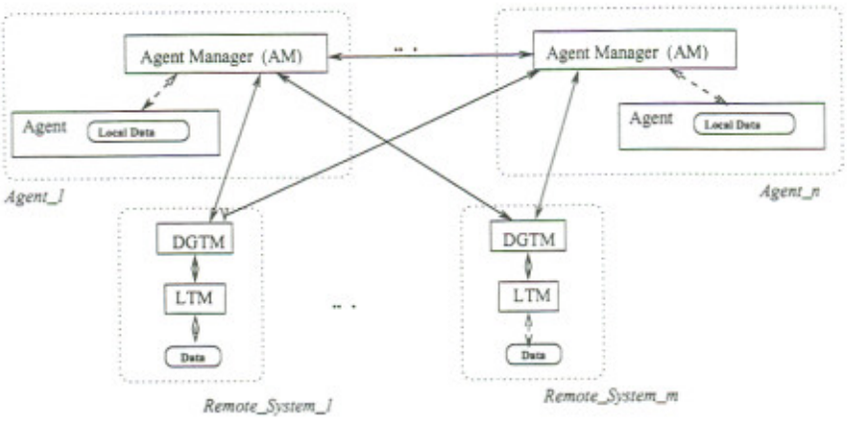

Figure 2: Agent execution

the corresponding remote system. The properties that an LTM ensures for the execution of a method vary based on the requirements of each system. For example, LTMs for database systems ensure the ACID properties of each basic method, that is, that each method is executed as an atomic, consistent, isolated and durable unit.

The above approach resembles traditional multidatabase transaction management techniques [BGMS92, PBE95]. The main difference is that there is no central point of control, in the form of a global transaction manager. Instead, control is distributed among agent managers and local schedulers (DGTMs). This way, bottlenecks that can seriously affect performance, especially in cases of widely distributed systems, are avoided. We outline how agents managers and DGTMs cooperate to ensure relatively serializable executions.

Upon its creation, each agent receives a *timestamp*. Each timestamp must be unique, for example, it can be defined to be a combination of the value of the clock and the user's *id*. The timestamp of an agent corresponds to its global serialization order. Each AM has two basic responsibilities. First, it coordinates the execution of its agent. It ensures that its agent execution is well structured. To enforce the specified structural dependencies, the *AM* can either use graph-based methods [BDG+94] or automata-based techniques [ASSR93]. Second, each *AM* produces correct relatively serializable executions on its local data consistent with

the timestamp order.

To handle breakpoints, the commutativity relation between the methods that follow the breakpoint and the methods specified in the breakpoint is changed at run time. In particular, the *AM* upon receipt of the *breakpoint* method, it modifies the commutativity relation so that for the duration of the breakpoint, the associated methods commute even if they normally do not. Delegation is taken into consideration directly in the definition of an agent execution by making a delegated method part of the execution of the agent it was delegated to. In particular, upon receipt of the *delegate* method, the *AM* treats the corresponding methods as part of its own agent execution.

Each DGTM produces DB_i correct relatively serializable schedules consistent with the timestamp order. We now describe the submission of a basic method from a DGTM to an LTM so that relatively serializability is ensured. To execute a composite method, each AM can use techniques such as the semantic-based locks of [MRW+93]. Each DGTM possesses a variable called a *logical ticket* (*LT*) and a list of the timestamps of all basic methods that have been submitted to the site. The logical ticket is the larger of the timestamps in the list. A method that does not commute with a submitted method is not allowed to execute concurrently with it; thus, if such a method arrives with a smaller timestamp than the timestamp of its conflicting method, it is aborted. Two commutable methods are executed concurrently without any further control.

Complications arise if we consider autonomous remote systems. In these systems, transactions may be executed outside the control of the DGTM by being directly submitted to the LTM. If we allow such submissions of autonomous operations directly to the LTM, indirect conflicts among commutable methods may arise through conflicts with autonomous operations; these can be avoided by forcing direct conflicts among them. In this case, an additional data item per site is needed. This data item is physically stored in that site and is called a *physical ticket* (*PT*) [GRS94]. All commutable methods submitted by a *DGTM* to the *LTM* read and

write the PT so that they become conflicting. This is accomplished by having each DGTM execute the following code after a commutable method M of an agent T is received [BRG92]:

```
get(LT)
if (LT > T's timestamp)
   abort(M)
else
   submit(M) to the LTM
   in a critical region
     get(LT)
     if (LT > T's timestamp)
         abort(M)
     else
         write(PT, T's timestamp)
            -- method M is executed --
         if decision taken to commit M
            set (LT, T's timestamp)
            commit(M)
         else abort(M)
```

□

In summary, each AM ensures that the execution of its agent is well structured. It also ensures that the projection of all operations on its agent's data is relatively serializable based on the timestamp order. Each $DGTM_i$ ensures that the execution of all basic methods at its corresponding site i is relatively serializable based on the timestamp order. Thus, schedule correctness is ensured.

The information that a method has been aborted or committed is passed from the LTM through the $DGTM$ to the corresponding agent manager that decides what the next action

will be. For example, in the case of a method being aborted, a compensating method may be submitted, or the aborted method may be retried. Similarly, if a prepare-to-commit state is supported by the system, this information is also passed from the LTM through the $DGTM$ to the AM. In this case, the AM can make its own decision on whether to commit or abort a method.

5 Related Work

The techniques for supporting consistency in mobile object models presented in this paper combine concepts from multidatabase concurrency control, advanced transaction models, and workflow management. [BGMS92] offers an excellent survey of the problem of concurrency control in multidatabase systems. However, the majority of multidatabase transaction management systems adopt a centralized approach; [WV90, BRG92] are possible exceptions. Many advanced transaction models have been proposed (see [Elm92] for examples). ACTA [CR94] provides a framework based on first-order logic for reasoning about extended transaction models. This model is low-level; a higher-level model based on transaction primitives is described in [BDG+94]. These two models can be used to express and implement respectively the structural characteristics of agents. On the basis of extended transaction models, many researchers have defined workflow specifications [RS95b, GHS95] similar to the agent structural dependencies. A very preliminary presentation of some ideas in this paper has appeared in [PB95]. In the current paper, we formalize these ideas in the form of a transaction model for agents and present protocols for the implementation of the model.

6 Conclusions

In this paper, we have presented a scheme for ensuring correctness of the concurrent execution of agents. The properties ascribe to the execution of an agent is that of advanced transactions. In particular, the structure of an agent is defined through dependencies between the execution

state of its methods. Furthermore, agents can cooperate with each other by sharing their intermediate results or by delegating the responsibility of specific actions to each other. The enforcement of the structural properties of an agent and the control of the interaction of agents with other agents and remote resources is assigned to per agent agent managers that accomplish this task through a well-defined small set of primitives. These primitives can be efficiently implemented in the form of library support for concurrency.

References

[ABAK94] D. Agrawal, J. Bruno, A. Abbadi, and V. Krishnaswamy. Relative Serializability: An Approach for Relaxing the Atomicity of Transactions. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems*, pages 139–149, 1994.

[ASSR93] P. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the 9th International Conference on Very Large Database Systems*, pages 134–144, 1993.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings of the 1994 SIGMOD Conference*, pages 44–54, May 1994.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of Multidatabase Transaction Management. *VLDB Journal*, 1(2):181–239, 1992.

[BHG87] P. A. Bernstein, V. Hadjilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addisson-Wesley, 1987.

[BRG92] P. K. Batra, M. Rusinkiewics, and D. Georgakopoulos. A Decentralized Deadlock-free Concurrency Control Method for Multidatabase Transactions. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, June 1992.

[CAC94] Special Issue on Intelligent Agents. *Communications of the ACM*, 37(7), July 1994.

[CGH+95] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 2(5), October 1995.

[CR94] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[Elm92] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[FO89] A. A. Farrag and M. T. Ozsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4):503–525, December 1989.

[GHS95] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2), 1995.

[GRS94] D. Georgakopoulos, M. Rusinkiewicz, and A. Sheth. Using Tickets to Enforce the Serializability of Multidatabase Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1), February 1994.

[HH91] T. Hadjilacos and V. Hadjilacos. Transaction Synchronization in Object Bases. *Journal of Computer and System Sciences*, 43:2–24, 1991.

[IEE97] Special Issue on Internet-based Agents. *IEEE Internet Computing*, 1(4), July-August 1997.

[Kle91] J. Klein. Advanced Rule Driven Transaction Management. In *Proceedings of the IEEE COMPCON*, 1991.

[MRW+93] P. Muth, T. C. Rakow, G. Weikum, P. Brossler, and C. Hasse. Semantic Concurrency Control in Object-Oriented Database Systems. In *Proceedings of the 9th International Conference on Data Engineering*, pages 233–242, 1993.

[PB95] E. Pitoura and B. Bhargava. A Framework for Providing Consistent and Recoverable Agent-Based Access to Heterogeneous Mobile Databases . *ACM SIGMOD Record*, 24(3):44–49, September 1995.

[PBE95] E. Pitoura, O. Bukhres, and A. Elmagarmid. Object-Orientation in Multidatabase Systems. *ACM Computing Surveys*, 27(2):141–195, June 1995.

[PS98] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*, volume 10 of *Advances in Databases Systems*. Kluwer Academic Publishers, 1998.

[RS95a] M. Rusinkiewicz and A. Sheth. Specification and Execution of Transactional Workflows. In W. Kim, editor, *Modern Database Systems*, pages 592–620. Addison Wesley, 1995.

[RS95b] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems*, pages 592–620. Morgan Kaufmann, 1995.

[VT97] J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet*. Springer Verlag, LNCS 1222, 1997.

[Whi96] J. E. White. Mobile Agents. General Magic White Paper, www.genmagic.com/agents, 1996.

[WV90] A. Wolski and J. Veijalainen. Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase. In *Proceedings of the Parbase90 Conference*, February 1990.