Transaction Oriented Computational Models for Multi-Agent Systems

Kotagiri Ramamohanarao
Department of Computer Science
University of Melbourne
Victoria, 3010, Australia
rao@cs.mu.oz.au

James Bailey
Department of Computer Science
University of Melbourne
Victoria, 3010, Australia
jbailey@cs.mu.oz.au

Paolo Busetta
ITC-IRST
via Sommarive 18,
38050 Povo, Trento, Italy
busetta@itc.it

Abstract

BDI (Belief, Desire, Intention) is a mature and commonly adopted architecture for Intelligent Agents. However, the current computational model adopted by BDI has a number of problems with concurrency control, recoverability and predictability. This has hindered the construction of agents having robust and predictable behaviour. Indeed the conceptual and practical tools needed for building dependable agent systems, resilient to faults and other unexpected situations, are still at an early stage of research.

To this end, we propose to use ideas from established database technology as an adjunct to BDI agent systems. In the long term, we envisage a two layer model for distributed system development. The upper layer, called "behavioural" in our scheme, consists of concepts and techniques coming from agent research. The lower layer, called "control", is based on more traditional computer technologies and implements the actions decided by the upper layer. The emphasis of the upper behavioural layer is on flexibility of the software development process, in order to be effective in application domains that are difficult to analyse or rapidly changing. By contrast, the main focus of the lower control layer is rigorous and efficient implementation within well-known boundaries.

We discuss the development of an agent system having a computational model with well-defined correctness criteria. Instead of hardwiring robustness and fault-tolerant behaviour into agent plans, well defined notions of correctness exist at the semantic level. Verification can then be undertaken at the desired level of abstraction.

1 Introduction

Intelligent agents are a very active area of AI research and possess very high potential for practical application in areas ranging from interactive information systems to active networks.

Of the various agent architectures which have been proposed, *BDI (Belief, Desire, Intention)* [13] is probably the most mature and has been adopted in a number of research and industrial applications. BDI agents are autonomous entities able to work in teams and react to changing environmental conditions. Although there is not yet a formal definition of an agent, researchers generally agree that for a computer program to be called a rational agent, it must be embedded in a dynamic and partially unpredictable environment and must show the following characteristics [16, 15]:

- *autonomy*: it operates without direct, continuous supervision;
- *social ability*: it is able to interact with other agents and possibly humans;
- *reactivity*: it has perceptions of the world inside which it is acting and reacts to changes in a timely, quasi real-time fashion;
- *pro-activeness*: its behaviour is not exclusively reactive but is also driven by internal goals, i.e. it may take initiatives.

The BDI architecture has been used in some products and a number of applications ranging from air traffic control to air combat simulations, from telephone call centres to the handling of malfunctions on NASA's Space shuttle [9]. The BDI approach is based on the study of *mental attitudes* [13] and tackles the problems arising when trying to use traditional planning in situations requiring real-time reactivity. The *Beliefs* represent the informational state of a BDI agent, that is, what it knows about itself and the world. *Desires* or *goals* are its motivational state, that is, what the agent is trying to achieve. A typical BDI agent has a so-called *procedural knowledge* constituted by a set of *Plans* which define sequences of actions and tests (*steps*) to be performed to achieve a certain goal or react to a specific situation. The *Intentions* represent the deliberative state of the agent, that is, which plans the agent has chosen for eventual execution.

In addition to the characteristics of autonomy, social ability, reactivity and pro-activeness, we argue that agents should possess two additional properties - namely *robustness* and *predictability*. However, the current computational model adopted by BDI has many problems concerning concurrency control and recoverability in general. This has hindered the construction of agents having robust and predictable behaviour. Indeed the conceptual and practical tools needed for building dependable agent systems, resilient to faults and other unexpected situations, are still at an early stage of research. More specifically, the architecture lacks a paradigm for concurrency control amongst intentions performing conflicting operations, such as trying to manipulate the same set of beliefs at the same time. In theory, this problem is resolvable by writing context-specific meta-plans policies. However, in addition to being impractical, writing meta plans which discover and handle race conditions in real-time is a very challenging task. There is also the potential of duplication of effort from a software engineering point of view. Another important feature missing in the BDI architecture is an exception handling mechanism. In particular, there is no prescribed way to report faults concerning the infrastructure or other agents in a team happening asynchronously with the execution of intentions. It is left to the application developer to provide adequate failure detection and recovery mechanisms.

It can also be argued that *the current BDI architecture lacks an adequate computational model expressing the activities an agent is performing at a given time and how they relate to one another*. Consequently, tasks such as moving an agent to another host or, analogously checkpointing to disk and recovering later are non trivial. This is because the world may change in the meantime and its new state must be taken into account by the reborn agent. Issues such as whether the beliefs should reconstituted by the agent and whether all goals and intentions should be moved become important. Neither the architecture nor the existing languages help in resolving these issues.

Furthermore, engineering large agent systems is a difficult task. So although the BDI architecture allows for multiple agents each having multiple threads of execution, the complexity of trying to design a multi-threaded, multi-agent program places high demands on the programmer. Indeed, anecdotal evidence suggests that instead of using these facilities, the programmer may often resort to a design which uses just a single agent having a single thread of execution.

To this end, we propose to use ideas from established database technology as an adjunct to BDI agent systems. In the long term, we envisage a two layer model for distributed system development (illustrated in figure 1). The upper layer, called "behavioural" in our scheme, consists of concepts and techniques coming from agent research. The lower layer, called "control", is based on more traditional computer technologies and implements the actions decided by the upper layer. The emphasis of the upper behavioural layer is on flexibility of the software development process, in order to be effective in application domains that are difficult to analyse or rapidly changing. By contrast, the main focus of the lower control layer is rigorous and efficient implementation within well-known boundaries. This can relieve the programmer from the burden of having to explicitly write code for handling concurrency and recovery.

Layer	Competence	Mechanism	Representative Technologies
Behavioural	Reaction/Deliberation Co-ordination/Negotiation	Mental attributes	Multi-agent frameworks BDI architectures
Control	Manipulation of information bases Sensing and action	Abstract data types Algorithms	OO programming Relational databases

Figure 1. A layered model of agent-based distributed systems

2 Background

2.1 The BDI Agent Architecture

Figure 2, extracted from [9] shows the basic components of a typical BDI agent. The *Beliefs* represent the informational state of the agent, that is, what it knows about itself and the world. *Desires* or *Goals* are its motivational state, that is, what the agent is trying to achieve. A typical BDI agent ([9, 12, 13] has a so-called procedural knowledge constituted by a set of *Plans* which define sequences of actions and tests *steps* to be performed to achieve a certain goal or react to a specific situation. The *Intentions* represent the deliberative state of the agent, that is, which plans the agent has chosen for eventual execution.

The agent reacts to *events*, which are generated by modifications to its beliefs, additions of new goals or messages arriving from the external world. An event may invoke (trigger) one or more plans; the agent commits to execute one or more of them, that is, they become intentions.

Intentions are executed one step at a time. A step can query or change the beliefs, perform actions on the external world, suspend the execution until a certain condition is met, and submit new goals. The operations performed by a step may generate new events which, in turn, may start new intentions.

An intention succeeds when all of its steps have been completed; it fails when certain conditions (either guarding its execution or being tested by a step) are not met, or actions being performed report errors, etc.

An agent applies a set of default policies when selecting which plans become intentions, how to schedule the active intentions, etc. These can be overridden by user defined policies, usually invoked via the same event/plan/intention mechanism described above (meta-level plan scheduling).

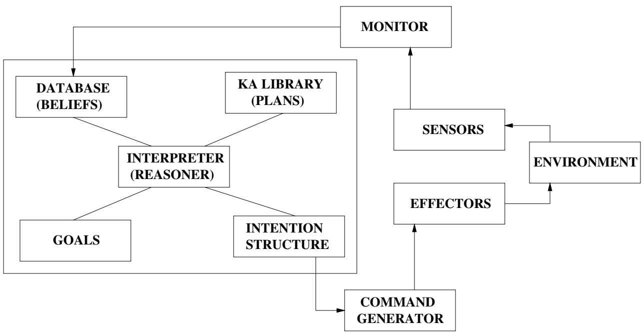

Figure 2. BDI Agent Structure

In summary, BDI is the abstract architecture of a family of parallel and distributed systems.

2.2 ACID transactions

We now discuss some concepts of *transaction processing* (TP). This description is based on [10, 11]. Traditional transaction processing systems prevent inconsistency and integrity problems by satisfying the so-called ACID *properties* of transactions: *Atomicity, Consistency, Isolation, Durability*.

Atomicity means that either all or none of transactions operations are performed.

Consistency means that each transaction has to maintain the integrity constraints on the objects it manipulates.

Isolation means that a transaction executes as if it were running alone, that is, without interference from concurrent transactions.

Durability means that all changes made by a successfully terminated *committed* transaction become permanent, surviving any subsequent failure.

In other words, the ACID properties define an abstract computational model in which each transaction runs as if it were alone and there were no failure. The programmer can focus on developing correct, consistent transactions, while the handling of concurrency and failure is delegated to the underlying engine.

2.3 Air-Traffic Example

We now describe a scenario which will be used as a running example in the remainder of the paper. The scenario being simulated is an airport with several runways and an air traffic controller agent looking after the arriving aircraft. An arriving aircraft notifies its current position and its earliest arrival time; it asks the controller when and on which runway to land. An aircraft may notify an emergency situation, requiring prompt allocation of the earliest possible time on a desired runway.

3 Nested Transactions

Nested transactions are an extension of ACID transactions. In a nutshell, the idea is to break down normal ("flat") ACID transactions into a main (*root*) transaction and a number of *subtransactions* (or *descendants*) invoked by the root; recursively, subtransactions can invoke other subtransactions. The set of transactions comprising a root and all its descendants forms a *family* (or descendants).

As a first step, we propose the use of nested transactions as a framework for the execution of plans. These can provide a framework for managing partial failures. Figure 3 shows an example of a nested transaction. By definition, a subtransaction doesn't have the durability of ACID transactions. If a subtransaction fails, all the actions that it and its descendants performed are aborted, but neither its siblings nor its parents are affected.

Figure 4 shows an example of a nested transaction inside a single agent. Returning to our air-traffic example: Suppose a plane enters the airport airspace and sends a message to the air traffic controller agent requesting a landing slot. In response to this event, the agent controller executes Plan A as a top level transaction with the goal of arranging a slot booking for this plane. Two of the sub-goals in this top level transaction are carried out as two subtransactions: Plan B in transaction 2 does the work of

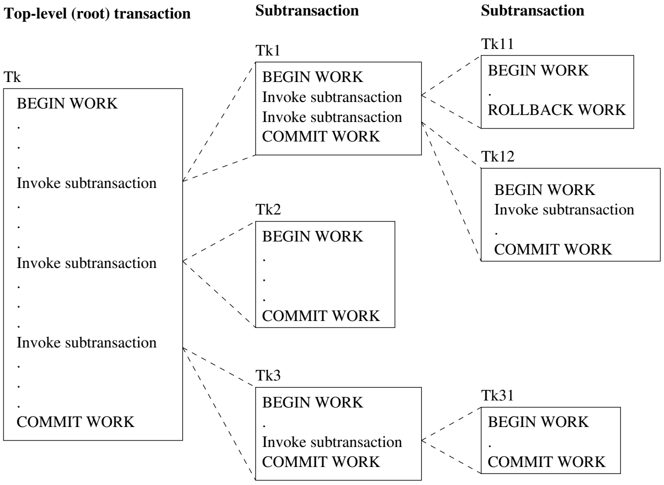

Figure 3. Nested Transaction Model

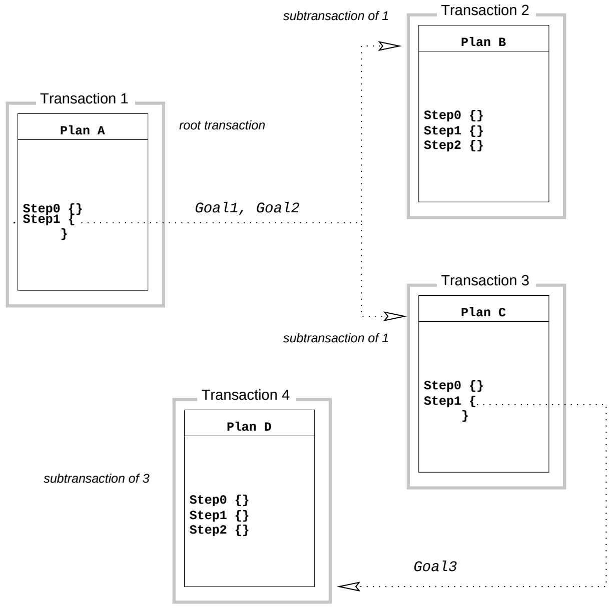

Figure 4. A Nested Transaction inside a single agent

scheduling a slot, by finding a free slot and then locking this entry in the data structure that records slot usage. Plan C in transaction 3 sends a message to the pilot of the aircraft and confirms whether he is willing to take the offered slot. Plan D in transaction 4 writes a record of the interaction between the controller and the pilot to a log file.

If Plan B fails (i.e. the agent is unable to find a free slot, perhaps because this plan only examines runways of length $<$ 1000m), this need not cause failure of transaction 1 as a whole. Instead, an alternative plan (e.g. Plan B' which searches for runways of length $\geq$ 1000m) could be launched as a subtransaction. This Plan B' may then be able to obtain a free slot.

Suppose Plan B succeeds, but plan C in transaction 3 fails (perhaps because the pilot decides that he doesn't need to land at this airport after all !). Suppose also that transaction 1 includes the constraint that if Plan C fails, then transaction 1 itself fails. Abort of transaction 1 would then cause rollback of Plan B, which has the effect of freeing up the slot entry which was locked earlier.

Observe that the agent may also be running other top-level transactions concurrently with transaction 1. These could be in response to events that are caused by the arrival of other aircraft that request runway slots. The handling of this concurrency is done automatically by the transaction manager. i.e. It is *not* necessary to explicitly write code that checks for the existence/behaviour/interference of other intentions (parallel threads). For example, there could exist two separate top level transactions holding locks on the slot data structure, one locking all slots with runways of length $<$ 1000m and the other locking all slots with runways of length $\geq$ 1000m. Neither of these transactions need explicitly be aware of the other.

4 Compensating Transactions

In the transaction scheme discussed in section 3, errors during transaction execution result in the transaction being "rolled" back to its initial state. A disadvantage of this is that the actions an agent takes cannot always be rolled back, since they may have affected the state of the real world and they can be no longer undone. In such a case, it may be inappropriate to model these as subtransactions and instead they should become top-level transactions, accompanied by compensating transactions whose job it is to perform something similar to 'rollback' if failure occurs.

Returning to our aircraft example from Section 3, let us impose the constraint that once a slot has been reserved (using Plan B), if confirmation is not received from the pilot within 3 minutes, then the pilot is assigned that slot regardless of his preference. Now consider what will happen under the nested transaction model if the pilot takes 5 minutes to respond to the confirmation request sent in Plan C. Plan C will fail and transaction 1 itself will fail (recall we have included the constraint that if plan C fails, transaction 1 as a whole fails). This would also cause rollback of Plan B in transaction 2, but this does not conform with our desired semantics.

The solution we propose is to allow the user to define an entity known as a compensating transaction. This has the effect of taking the agent to a state where various constraints are guaranteed to hold. These constraints may depend on the goals the agent possesses.

Thus, each compensatory transaction T^* is associated with another transaction T and it provides logic to undo (or alternatively compensate for) the actions of T, if T has failed, otherwise it does nothing. By definition a compensating transaction itself never fails.

Returning to our example, we remodel the logic as the top level transaction sequence $T_x; T_y;$. T_x is a transaction that executes Plan B (find and reserve a slot) and is followed by T_y, which sequentially executes two subtransactions T_z and T_x^*. T_z executes Plan C (confirm with pilot), whereas T_x^* is a compensating transaction for plan B. T_x^* has the logic

T_x^* : if received_confirmation or response_time ≥ 3 then $\{\}$ else free(slot_reserved_by_Plan_B);

The key point is that rollback of Plan B no longer automatically occurs when (conceptually) Plan A fails (as in the non-conforming example just highlighted). Rather, it is conditional on the response time of the pilot, as desired.

Thus, compensating transactions allow additional flexibility in modelling dependencies between failures of actions. Of course, compensating transactions may not exist for some types of actions. Therefore, a key question is - for what types of transactions is it possible to define associated compensating transactions ? See [7] for a discussion of this from a database perspective. Furthermore, if compensating actions are possible, how should one remodel a nested transaction as several top level transactions plus associated compensating transactions ? The next section describes one possible technique.

5 Transaction Chopping

Modelling the execution of plans within (sub)transactions can give rise to another difficulty in addition to the challenge of rollback for certain types of actions. The difficulty arises because locks taken on objects need to be held for the duration of the transaction (in case the transaction itself should fail).

The consequences are:

- For long-running transactions, keeping such locks for the duration of the transaction can result in more contention for resources (from other transactions waiting to access them).

- Execution of an entire plan within a transaction may not be sufficiently responsive in a dynamically changing world, since the effects of the transaction may not be visible until the transaction has completed.

A way to address these issues that increases performance and also increases interactivity with the outside world, is to use a technique known as *transaction chopping* [14]. Chopping transactions is a well known idea in database systems theory for shortening transactions. A single (longer) transaction is automatically decomposed into a series of smaller transactions. These smaller transactions may run concurrently, thus increasing performance and potentially separating out problematic interactions into different transactions.

For example, let us consider a reworking of the running air-traffic example. Suppose a plane enters the airport airspace and sends a message to the air traffic controller agent requesting a landing slot. In response to this event, the agent controller executes a top level transaction T=t1;t2 where t2 is a subtransaction that finds and confirms a slot with the pilot and t1 is a subtransaction that assigns a docking bay for the plane for when it finally lands. In this case, t2 cannot complete until t1 has completed. So, if execution of t1 is subject to a lengthy delay, then the pilot will not be given a confirmed landing slot in a timely manner. Of course the solution in this case is to chop t1 and t2 up into separate top-level transactions, which may run independently and concurrently. Techniques for transaction chopping can provide an automated technique for achieving this [14].

Transaction chopping also has potential in the production of compensatory plans. A desirable goal would be if the user could specify points within a transaction where compensation may be required. The system would then automatically chop up the original transactions and merge them with some additional compensating transactions.

6 Active databases

It has been observed that agent systems have much in common with the technology of *active databases* [3]. These are databases containing rules which can evaluate conditions and perform actions in response to events. Such rules execute within a well-defined transaction context. Another direction of research includes the integration of active database technology with agent technology. The intention being, to identify features of an agent language for which it possible to do a translation into the “lower” level form of active rules, while retaining the desired semantic correctness properties. We can then employ methods for active rule analysis and optimisation [1, 2, 5, 6, 4] to assist with analysis of the higher level agent(s).

7 Related and Future Work

To our knowledge, there is no previous work on designing a language for BDI agents with a transactional/database flavour. This is because, to a significant extent, the database and AI communities have tended to operate independently. Indeed, research in AI has tended to emphasise exciting features such as the "intelligence" of agents, whilst work on aspects such as recoverability and predictability has lagged behind. On the database side, there has been much work on extending transactional models to cater for workflow applications. This is certainly relevant to our aims, but doesn't directly address the models used by BDI agents. We hope that the line of work contained in this paper will help promote cross fertilisation of results between the two disciplines.

In previous work [8], a first attempt was undertaken on the development of a transaction oriented multi-agent system (TOMAS). This resulted in a proof of concept system using a BDI agent architecture with a nested transaction model. It didn't, however, have a specific programming language associated with it, nor tools for system analysis and development. Consequently, an area of further investigation is to extend these ideas by designing and implementing an agent language based on extended transactional notions of correctness (drawn from database system theory) and exhibiting properties of reliability and predictability.

In other relevant work [3], we identified a correspondence between BDI agents and active databases. These are databases containing rules which can evaluate conditions and perform actions in response to events. Similar to agent systems, methods of analysing the behaviour of active rules are highly desirable and in a series of papers we have presented fundamental results on rule analysis, ranging from decidability issues to approximate methods based on abstract interpretation [1, 2, 5, 6, 4].

8 Summary

We have described the use of transactional notions for designing BDI agents. Transactions are well understood in the database community and possess ACID properties which help the designer understand the semantics of execution. Since well-defined correctness notions exist at the semantic level, it is not necessary to hardwire robust and fault tolerant behaviour into agent plans.

Nested transactions appear to provide a good facility for the execution of agent plans. For situations where transaction granularity causes a loss of flexibility/efficiency/responsiveness, we have discussed the use of compensatory plans and transaction chopping.

References

[1] J. Bailey, G. Dong, and K. Ramamohanarao. Structural issues in active rule systems. In *Proceedings of the Sixth International Conference on Database Theory, LNCS 1186*, pages 203–214, Delphi, Greece, 1997.

[2] J. Bailey, G. Dong, and K. Ramamohanarao. Decidability and undecidability results for the termination problem of active database rules. In *Proceedings of the 17th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 264–273, Seattle, Washington, 1998.

[3] J. Bailey, M. Georgeff, D. Kemp, D. Kinny, and K. Ramamohanarao. Active databases and agent systems - a comparison. In *Proceedings of the Second International Workshop on Rules in Database Systems, LNCS 985*, pages 342–356, Athens, Greece, 1995.

[4] J. Bailey and S. Mikulas. Expressiveness issues and decision problems for active database rules. In *Proceedings of the 8th International Conference on Database Theory LNCS 1973*, pages 68–82, London, 2001.

[5] J. Bailey and A. Poulovassilis. An abstract interpretation framework for termination analysis of active rules. In *Proceedings of the 7th International Workshop on Database Programming Languages LNCS 1949*, pages 249–266, Kinloch Rannoch, Scotland, 1999.

[6] J. Bailey and A. Poulovassilis. A dynamic approach to termination analysis for active database rules. In *Lecture notes in Computer Science 1861. Proceedings of the 1st International Conference on Computational Logic (DOOD stream)*, London, 2000.

[7] P. Bernstein and E. Newcomer. *Principles of transaction processing for the systems professional*. Morgan Kaufmann, 1996.

[8] P. Busetta and K. R. An architecture for mobile BDI agents. In J. Carroll, G. B. Lamont, D. Oppenheim, K. M. George, and B. Bryant, editors, *Proceeding of the 1998 ACM Symposium on Applied Computing (SAC'98)*. ACM Press, 27 February - 1 March 1998. An extended version appears as Technical Report 97/16, Department of Computer Science, The University of Melbourne, Australia, 1997.

[9] M. P. Georgeff and F. F. Ingrand. Decision - making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Detroit, Mi., USA, 1989.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[11] K. Ramamritham and P. K. Chrysanthis. *Advances in Concurrency Control and Transaction Processing - An Executive Briefing*. IEEE Computer Society Press, 1996.

[12] A. S. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *MAAMAW'96: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, LNAI 1038. Springer-Verlag, Jan. 1996.

[13] A. S. Rao and M. P. Georgeff. An Abstract Architecture for Rational Agents. In W. S. C. Rich and B. Nebel, editors, *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[14] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3):325–363, 1995.

[15] Y. Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[16] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(12):115–152, 1995.