A Reliable Computational Model For BDI Agents

Paolo Busetta
ITC-irst
38050 Povo, Trento - Italy
busetta@itc.it

James Bailey and
Kotagiri Ramamohanarao
Department of Computer Science
University of Melbourne
Victoria, 3010, Australia
{jbailey,rao}@cs.mu.oz.au

General Terms
Reliability, Design

Keywords
BDI, multi-agent systems, ACID transactions, nested transactions

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.

To this end, we propose to integrate *distributed transactions*, a well-established technology in distributed systems, into the computational model of multi-agent systems based on the BDI architecture. Differently from common approaches, where so-called ACID (Atomic, Consistent, Isolated, Durable) transactions are used simply to operate on external resources such as databases, in our model transactions are the foundation of the operational semantics of intentions and of collaborative tasks within team of agents. They provide a predictable, well understood behaviour in case of partial or total failure of intentions to achieve their goals or even crashes of agents. Furthermore, distributed transactions provide a simple and clear extension of the BDI semantics from the single-agent case to teams of agents.

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.

Two BDI interpreter prototypes have been developed to demonstrate the feasibility of our approach. The first, TOMAS, is a Java environment that execute intentions as *nested transactions*. The second is a re-implementation of TOMAS within a J2EE application server, which can be used to develop session beans (i.e., business logic); it demonstrates how the model we propose nicely fits into a state-of-the-art environment for mission critical systems in domains such as e-business and Web services.

1. INTRODUCTION
Of the various agent architectures which have been proposed, *BDI (Belief, Desire, Intention)* [15] is probably the most mature and has been adopted in a number of research and industrial applications.

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 [10]. The BDI approach is based on the study of *mental attitudes* [15] 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 which are commonly indicated as distictive of intelligent agents (autonomy, social ability, reactivity and pro-activeness [19, 18]) 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.

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* [15]. 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 system places high demands on the programmer. Indeed, anecdotal evidence suggests that instead of using these facilities to their full power, programmers often resort to a programming style that closely resembles rule writing for traditional ECA (Event-Condition-Action) systems, where a single rule at the time is scheduled and executed atomically. Of course, such a style dramatically simplifies the analysis of the behaviour of an agent, which otherwise – lacking an adequate computational model in particular concerning concurrency, as said above – becomes hard to understand and control.

To overcome the limitations mentioned above, we propose to integrate *distributed transactions*, a well-established technology in databases and distributed systems, into the computational model of multi-agent systems based on the BDI architecture. Differently from common approaches, where ACID transactions (described in Sec. 2.2 below) are used simply to operate on external resources such as databases or tuple spaces, in our model transactions are the foundation of the operational semantics of intentions and of collaborative tasks within team of agents. This provides a predictable, well understood behaviour in the face of partial or total failure of intentions to achieve their goals, or even crashes of agents. Furthermore, distributed transactions provide a simple and clear extension of the BDI semantics from the single-agent case to a teams of agents.

This paper is organized as follows. Next section provides background information on BDI and ACID transactions, and introduces a running example used throughout the rest of the paper. Sec. 3 introduces the core of our approach, i.e. the integration of nested ACID transactions with BDI intentions; Sec. 4 discusses its extension from the single to the multi-agent case. ACID transactions are powerful but have limitations, mostly concerning long-term activities; Secs. 5 and 6 introduce alternative transaction models that may be adopted in combination with BDI. A comparison between BDI and active databases is provided in Sec. 7. Sec. 8 illustrates our current implementation of transaction-based BDI systems. Finally, we draw some comparison with works available in the liturature in Sec. 9.

2. BACKGROUND

2.1 The BDI Agent Architecture

Figure 1, extracted from [10], 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 ([10, 14, 15] has a so-called procedural knowledge constituted by a set of *Plans* which define sequences of action and test *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.

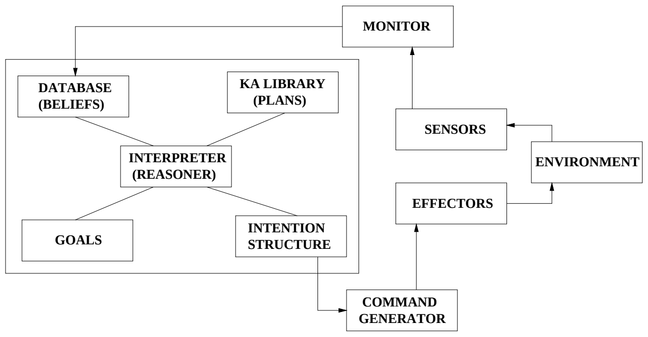

Figure 1: BDI Agent Structure

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). Particularly important is the case of intention completion; at that time, the agent reconsiders the reason that caused it to commit to the intention (that is, the agent examines the triggering event). In the case of a goal, default policy is to consider success as a sign that the goal has been achieved; on failure, standard policy is to select and execute another plan until all alternatives have been attempted, at which point the goal is considered as unachievable. This automatic retry-on-failure mechanism may help in dealing with changes in the environment during execution, lack of accurate information at the time of plan selection, and similar cases.

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 [11, 13]. 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.

ACID transactions are supported by most, if not all, databases, including all commercial relational databases we are aware of. Isolation is commonly guaranteed by means of *locks*, while atomicity and durability are managed by *logging* the operations performed by a transaction and applying appropriate recovery or roll-back actions in the event of a failure.

ACID and nested transactions, as well as various other transaction models (including those described later), are also commonly adopted in distributed systems to achieve reliable behaviour. For instance, many state-of-the-art J2EE-compliant[1] distributed transaction systems by major software vendors provide JTA/JTS-compliant[2] transaction management. J2EE platforms commonly coordinate multiple databases and application servers. It also worth noticing that some tuple-based communication platforms often used for agents, such as TSpaces or JavaSpaces, provide transactions for safe tuple manipulation (a good overview on the matter can be found in [16]).

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 AS SEMANTICS OF INTENTIONS

Nested transactions are an extension of ACID transactions. In a nutshell (based, as above, on [11, 13]), 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.

[1]Java^{TM}2 Enterprise Edition, `http://java.sun.com/j2ee/`
[2]Java Transaction API and Service, `http://java.sun.com/j2ee/transactions.html`

The set of transactions comprising a root and all its descendants forms a *family* ((or *tree*).

When a subtransaction commits (i.e., terminates successfully), its effects do not become durable but are simply added to those of its parent; simply put, the subtransaction “merges” with its parent. This applies recursively up to the root transaction, whose final commit is the only one that makes all changes durable. A subtransaction can fail, causing all the work it and its descendants did to be rolled back, i.e. all resources are returned to their original state; however, neither siblings nor its parent are affected, and the latter can take recovery actions, including running alternative subtransactions.

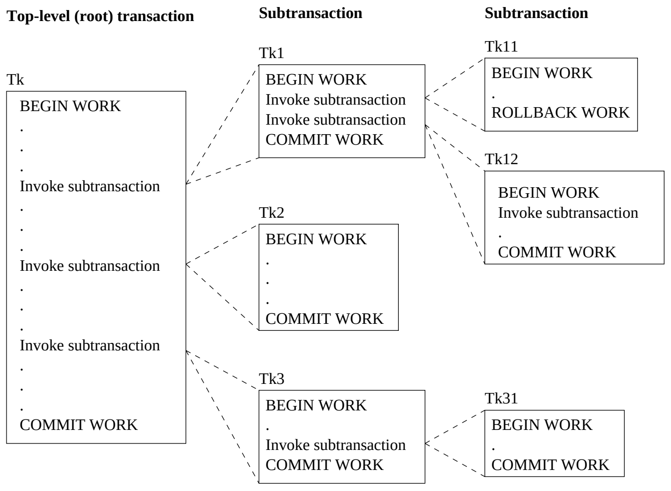

Figure 2: Nested Transaction Model

As a first step, we propose the use of nested transactions as a framework for the execution of plans, i.e. as the underlying semantics of intentions. Nested transactions, combined with the automatic retry-on-failure mechanism for goals described in Sec. 2.1, can provide a flexible framework for managing partial failures. Figure 2 shows an example of a nested transaction. When subtransaction Tk11 fails, all the actions that it performed are aborted, but neither its sibling Tk12 nor its parent Tk1 are affected. It is worth to stress that “actions” include any manipulation on resources, including those internal to an agent such as its own beliefs.

Figure 3 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 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 and these have all been allocated), this need not cause failure of transaction 1 as a whole. Instead, an alternative plan (e.g. Plan B’ which searches for free runways of length ≥ 1000m) could be launched

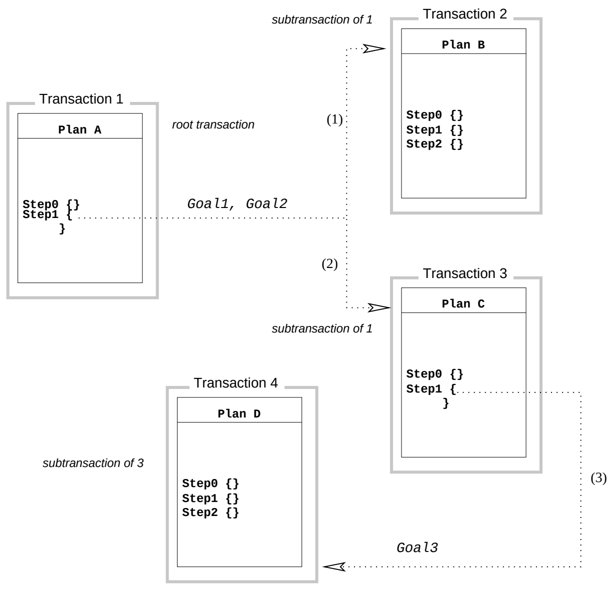

Figure 3: A Nested Transaction inside a single agent

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 engine of the agent managing the transactions; 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.

Transactionality as semantics for intentions gives also some significant advantage in terms of meta-level programming. Consider, for instance, that intentions can be safely aborted and resubmitted, as often done in transactional systems on deadlock detection; thus, an agent can apply this technique when faced with emergencies or other situations that force the execution of some higher priority activity. A similar mechanism has been proposed for agent mobility [9].

4. NESTED TRANSACTIONS FOR TEAMS OF AGENTS

As mentioned in Sec. 2.2, ACID transactions are commonly used to coordinate distributed systems. Propagation mechanisms that extend transactions from one process to another are normally built into the communication protocols, so that all systems involved in a common distributed activity participate to a single transaction. So-called *transaction monitors* are in charge of coordinating all systems involved in a transaction when commiting or rolling back; to this end, the well-known Two Phase Commit protocol [11] is commonly adopted and implemented by popular distributed system platforms such as Jini[3]. Transaction monitors may be distributed themselves, i.e. more than one may be involved in managing a single transaction.

We apply the same approach, and extend the transaction-based semantics of intentions described in Sec. 3 by augmenting each message sent by an agent with information on the transaction being executed by the sending intention. The receiver becomes partner of the caller in managing the transaction, which means that any intention handling the message becomes a sub-transaction of the sender's. In other words, the nested transaction family is propagated to both agents; further messages extend the nested transaction tree to other agents. The effect of this – apparently simple – extension is far-reaching, both theoretically and practically. Indeed, the intentions of all members of a team directed towards a common objective are now related by a common distributed nested transaction family. The same concurrency control and reliability characteristics of transactions within a single agent mentioned in Sec. 3 are now extended to all agents participating to concurrent distributed computations.

A couple of notable implications are worth highlighting. First, if a message is not handled successfully by its receiver, the sender's transaction fails, in turn causing the sender's intention to fail and eventually the BDI retry-on-failure mechanism to trigger. This behaviour automatically catches a large class of problems that otherwise should be handled explicitly, saving many acknowledgements and synchronization actions: for instance, the receiver of a message may not know how to handle it, or its sub-goals might not be achievable – possibly because of the failure of other agents –, or it may crash before the transaction commits.

The second important implication derives directly from the nested transaction model, as explained in Sec. 3: when a sub-transaction commits, its effects do not become durable but are simply added to those of its parent, until the root eventually commits. A consequence of this model is that a crash of an agent any time *after* it successfully reacted to a message, but *before* the root transaction commits, causes a partial failure that affects the branch of the family in which it was involved up to the highest not-yet-committed sub-transaction – possibly the root itself, i.e. the entire transaction family.

The example in Fig. 4 shows a transaction family spanning three agents. In our air-traffic control example, A_1 may be an arriving aircraft that asks for a landing time and a runway to air-traffic controller A_2; in turn, this delegates the runway scheduler A_3 of taking a decision; eventually, A_3 contacts aircraft A_1 to notify it of the allocated slot.

The root transaction, T_1, has been created for the intention executing plan P_1, attempting to achieve the high-level team objective (letting the aircraft to land). Observe how sub-transaction T_2, looking after plan P_2 (whose goal is to contact the air-traffic controller)

[3]In Jini, the transaction monitor is called *transaction manager*. See the "Jini Technology Core Platform Specification", `http://wwws.sun.com/software/jini/specs/`

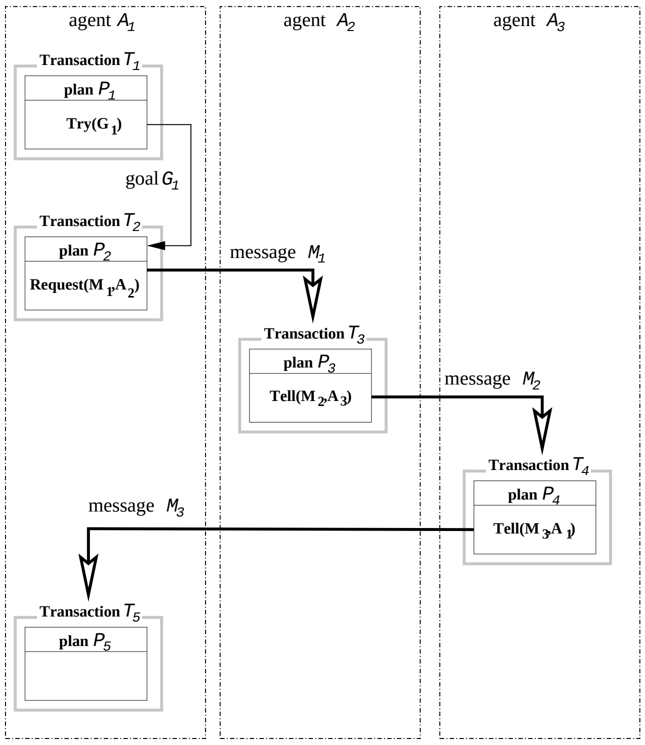

Figure 4: A Nested Transaction in a Team

is extended by message M_1 to A_2; its descendent T_3 is created for the intention reacting to M_1. The same happens with T_3, extended by M_2, and T_4, extended by M_3. T_5, which runs for P_5 (reacting to the scheduling notification), is particularly interesting since it is executed by the same agent that is controlling the root; a failure of plan P_4 by A_3 mid-way its transaction (for instance, because an emergency landing has been required) will cause the effect of T_5 to be rolled back, but without affecting the rest of the work of A_1.

Observe that simultaneous allocation requests to the controller from different aircrafts are automatically serialized whenever they may need to modify the same beliefs or data structures (most likely to happen within A_3). An emergency request may be handled by A_2 by forcing the failure of all its intentions executing the same plan P_3 (this may be easily obtained with an appropriate guard on the plan, the so-called *maintenance condition*), thus forcing all on-going scheduling to be rolled back, and letting an emergency-handling intention running at high priority while re-submitting all aborted scheduling requests.

Another interesting failure case to analyze is a communication break down between the aircraft A_1 and the controller A_2 while the transaction is on-going. Since the distributed commit necessarily fails, all effects of plans P_2 to P_5 are automatically rolled back; P_1 now has to handle the failure in getting a runway and behave consequently (perhaps simply by resubmitting the same request after a while).

As shown above, the distributed nested transaction model is powerful, but clearly it is not directly applicable to all types of distributed computations, such as anything requiring long-term collaborations. To this end, our initial prototype, TOMAS (Sec. 8), allowed the agent to choose which transaction policy to apply for each intention by means of a simple meta-level decision. Moreover, TOMAS allowed selected beliefs not to be treated as resources under transaction control, and supported two different types of messaging, one propagating and the other not propagating transactions. Before discussing our current implementations, we illustrate a couple of transaction models that tackle some of the inflexibilities of ACID transactions.

5. 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, by revising the effect of the committed transaction.

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 should itself never fail.

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 following logic:
T_x^* : if received_confirmation or response_time $\geq$ 3 then $\{$null$\}$
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, an important open area of research is to investigate the types of transactions for which it is possible to define associated compensating transactions. Work in [7] discusses this from a database perspective. Another important question is to determine how a nested transaction may be remodelled as several top level transactions, plus associated compensating transactions. The next section describes one possible technique.

6. 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* [17]. 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 [17].

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.

7. ACTIVE DATABASES

Another important facility of database systems that relates to agents, is the use of rules which can evaluate conditions and perform actions in response to events. Such rules execute within a well-defined transaction context. Databases with such rules are called *active databases* [3]. An interesting direction of research therefore aims at 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).

8. TRANSACTION-BASED, MULTI-AGENT BDI PLATFORMS

In previous work [9] (expanded and more formally described in [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 spanning teams of cooperative agents. In TOMAS, every agent acted as monitor of its own transactions. Isolation of operations on beliefs were guaranteed via a locking mechanism. Two types of actions were available for messaging: *send* and *post*. Sent messages were extended with transaction information, so that distributed nested transactions were supported by a truly distributed transaction management; the Two-Phase Commit protocol ran every time an intention finished its execution. By contrast, posted messages acted on transactional queues (similar to those compliant with the JMS specifications[4]), that means that a message was actually sent only if and when its posting transaction committed.

At the core of TOMAS, a meta-level framework allowed the programmer to select the underlying semantics (called *policy*) for intention execution; plain, non-recoverable as well as nested ACID transaction policies were analyzed and prototyped. TOMAS 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.

TOMAS has been recently revised for a J2EE (Java 2 Enterprise Edition) environment. This environment is particularly suitable for robust agent systems, due to its automatic support for messaging and transactions.

Within the environment, agents act as an intelligent session Enterprise Java Beans. Under the J2EE model, a session bean is a component containing logic associated with a particular client session or task. Programmers may create these intelligent session beans by extending an existing agentbean class and providing the agent with a set of initial beliefs, goals and plans. Agents run on a J2EE application server, which automatically provides both messaging via JMS and transaction functionality. This design is a natural extension of native J2EE features and greatly simplifies the original structure of TOMAS. The BDI-agent adds "intelligence" in the sense of rule-based and intention-based behaviour, with the agents providing the session bean application logic. Possible application domains for this design include all e-commerce, web services, any kind of Internet or wide-area network based control system including many telecom services.

9. 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

[4]Java Message Services, `http://java.sun.com/products/jms/`

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.

ACID and compensatory transactions are used in combination with agents in the work presented in [12]. In this setting, a *planning agent* creates *transaction trees*, possibly in cooperation with other planning agents when working on a joint goal. A tree is composed of simple agent actions, each encapsulated within its own an ACID transaction. A tree contains control parameters and control flow rules connecting its actions; these may include also *contingency* actions, i.e. compensatory transactions to be taken in the event of a failure of the plan to achieve its original goal or one of its subgoals. A tree is passed to an *execution agent*, which performs its actions by traversing the tree in the order specified by its controls. The system has been applied to distributed database environments, and the actions perfomed by agents consists of queries and updates.

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 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].

10. CONCLUSIONS

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, or responsiveness, we have discussed the use of compensatory plans and transaction chopping.

11. 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. A transaction based multi-agent architecture. Master's thesis, Department of Computer Science, University of Melbourne, 1999.

[9] P. Busetta and Kotagiri 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.

[10] 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.

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

[12] Khaled Nagi. Modeling and Simulation of Cooperative Multi-Agents in Transactional Database Environments. In *Proceedings of the Second workshop on Infrastructure for agents, multi-agent systems and scalable multi-agent systems, at the Fifth International Conference on Autonomous Agents (Agents2001), Montreal, Canada*, June 2001.

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

[14] Anand 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, January 1996.

[15] Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In W. Swartout 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.

[16] Davide Rossi, Giacomo Cabri, and Enrico Denti. Tuple-based technologies for coordination. In Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 83–109. Springer-Verlag, 2001.

[17] 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.

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

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