An Architecture for Mobile BDI Agents

Paolo Busetta    Kotagiri Ramamohanarao

Technical Report 97/16

Department of Computer Science
The University of Melbourne
Parkville, Victoria 3052
Australia

Abstract

BDI (Belief, Desire, Intention) is a mature and commonly adopted architecture for Intelligent Agents. BDI Agents are autonomous entities able to work in teams and react to changing environmental conditions. However, the current computational model adopted by BDI has problems which, amongst other limitations, prevent the development of mobile agents. In this paper, we discuss an architecture, TOMAS (Transaction Oriented Multi Agent System), that addresses these issues by combining BDI and the distributed nested transaction paradigms. An algorithm is presented which enable agents in TOMAS to become mobile.
1 Introduction

Intelligent Agents are a very active area of AI research [WJ95] [Sho93]. Of the various agent architectures which have been proposed, BDI (Belief, Desire, Intention) [RG92] is probably the most mature and has been adopted by a few industrial applications. BDI Agents are autonomous entities able to work in teams and react to changing environmental conditions.

Mobile multi-agent BDI systems would have many advantages. For instance, they would be able to

- perform tasks closer to sources of data;
- leave and join teams;
- balance the load in a network of computers.

However, the current computational model adopted by BDI has many problems concerning concurrency control and recoverability in general; these limitations prevent the development of mobile BDI agents.

In this paper, we propose an architecture, TOMAS (Transaction Oriented Multi Agent System), that addresses these issues. TOMAS has been inspired by previous work [BGK+95], and combines computational aspects of BDI with distributed nested transactions.

The nature of the problems discussed in this paper is the same of those faced by any distributed system. The model being proposed to tackle these problems is very well known and has been applied in several other situations. In our opinion, it will become even more common in future. Combining Intelligent Agents and transaction processing can help in making AI technology a part of the Software Engineering mainstream.

Section 2 is an overview of intelligent agents and BDI in particular. A few situations where mobile intelligent agents would be advantageous are presented in section 3. Section 4 discusses what prevents the current BDI agents from being mobile. Sections 5 and 6 present an extensible BDI toolkit written in Java and a prototype of TOMAS built by using the toolkit. Section 7 discusses how the computational model of TOMAS enables the mobility of agents. Finally, sections 8 and 9 briefly compare TOMAS with other systems and present future research directions.

2 The BDI Agent Architecture

The AI community has been researching in the field of intelligent (or rational) agents for more than a decade. There is not yet an authoritative definition of an intelligent agent, since different goals and environments have led to the development of a variety of languages and architectures. However, 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 ([WJ95], [Sho93], [BGK+95]):

- **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;
- **proactiveness**: its behaviour is not exclusively reactive but it is also driven by internal goals, i.e., it may take initiatives.

An approach to the study of rational agency which has received a great deal of attention is the so-called *Belief, Desire, Intention (BDI)* architecture. The BDI architecture has been used in some products (for instance, [Aus96]) 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.

The BDI approach is based on the study of *mental attitudes* [RG92] and tackles the problems arising when trying to use traditional planning in situations requiring real-time reactivity (see, for instance, the discussion in [GI89]). Figure 1, extracted from [GI89], shows the basic components of a typical BDI agent.

---

**Figure 1**: Procedural Reasoning System (PRS): System structure
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 [GI89], [RG92], [Rao96] 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 the 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 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 planning).

In recent years, a lot of research has been done in the area of collaborative work for teams of BDI agents [KLR+94]. The architecture has been extended to manage communication in a way largely inspired by the paradigms of the speech acts theory [Sea70]: messages are qualified as informational (tell) or directional (request, reply). In multi-agents systems, the members can have very complex relationships with each other concerning the organization of the team, their shared goals and beliefs, and their reciprocal cooperation in order to accomplish tasks.

In summary, BDI is the abstract architecture of a family of parallel and distributed systems working alone or in a team in dynamic environments. BDI allows a high degree of sophistication and sensitivity to the context when deciding how to react to changed conditions. Depending on design and implementation choices, BDI agents can show very different levels of reactive (event driven) and planned (goal driven) behaviour. The description of this behaviour is done in cognitive terms, i.e., by attributing mental attitudes. Research is being undertaken in a number of areas of relevance both to computer and cognitive sciences, for instance cooperative work and social commitments and recognition of intentions [Rao94].
3 Mobility and BDI Agents

At the time of writing, we are not aware of any specific work in the area of mobility of BDI agents. This is surprising when considering the level of maturity of the field of mobile agents, with industrial standards being published (e.g., [Gro97]) and available products. Multi-agent systems could benefit from integration with mobility services in many ways; for example,

- Teams could be formed by a mixture of “heavyweight”, cognitively complex agents dispatching tasks to lightweight agents pursuing only a few goals. Those lightweight agents could leave the team, move themselves to other hosts and possibly rejoin the team when they have accomplished their goals.

  There would be a number of benefits from such a team organization. For instance, efficiency can be improved by moving lightweight agents performing queries over a large database to the host of the database itself. Security and privacy restrictions could be satisfied more easily when collecting anonymous statistics on sensitive information, because only the results of the queries but no primitive data would be transferred. Integration with legacy systems would be eased by building lightweight agents acting as flexible bridges. Response time and availability would improve when performing interactions over network links subject to long delays or interruptions of service (e.g., controlling a robot on Mars from the Earth or interacting with a human via a portable notebook).

- A variation on the previous scenario would see a complex agent cloning itself in order to accomplish a specific goal on a specific host. Such a capability is particularly interesting when an agent makes sporadic use of a valuable shared resource (e.g., special purpose user interfaces, high speed graphical engines, etc.).

- Mobile BDI agents would allow the balancing of the computational load among hosts. They would make it easier to build highly available services and would improve the overall robustness of a multi-agent system.

- Dynamic loading of plans on never-stopping agents, distribution of agent software, services such as pay-per-use: all would benefit from the infrastructure provided for mobile agents.

4 Shortcomings of current BDI

Unfortunately, the existing BDI systems have a number of shortcomings preventing mobility of agents.
The architecture lacks a paradigm for concurrency control among 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-level policies. However, in addition to being impractical, writing meta-plans which discover and handle race conditions in real-time is a very challenging task.

The problems with concurrency are not perceived in some implementations thanks to a number of clever choices made by the system designers. For instance, in spite of their inherent parallelism, BDI agents are usually described and implemented as sequential machines executing exactly one step of all the active intentions at a time [RG92], [Rao96]. The justification for this is that it allows an analysis of the agents’ behaviour which otherwise would seem chaotic. Also, the default meta-level policies applied by an agent are extremely simple but very restrictive. In a typical system [GI89] [Aus96], each intention is given a priority and intentions of the same priority are ordered with a LIFO or FIFO criteria for scheduling purposes. The scheduler picks the first executable intention at the highest priority level and runs it until it terminates or suspends itself. Finally, many agents have a mostly reactive behaviour and perform only short, fast plans. By combining sequentialization of steps, default meta-level policies and short plans, the result is that most intentions run as atomic units, thereby avoiding concurrency problems.

In spite of all these precautions at the architecture and system design levels, it is extremely easy to build an agent suffering from race conditions. As we commented above, it is not easy at all to write policies preventing these conditions. The developer of an agent has to keep all of these concurrency issues in mind while writing an application, adding further cognitive burden to the design and implementation processes.

Pragmatically, workarounds boil down to very strict, ad hoc engineering disciplines which impose limitations on the exploitation of parallelism in the system. Such disciplines are hard to enforce automatically, let alone proving the robustness of the final software. In these contexts, scaling an agent from a small to a large application — in terms of the variety of situations it can manage and the complexity of its behaviour — becomes extremely risky even if the run-time load (that is, the number of intentions pursued concurrently) is maintained light. It is always possible to inadvertently break subtle assumptions about, for instance, the order in which events are received and managed and the way intentions are selected and executed.

It is our opinion that none of the issues concerning concurrency and agents is different in any essential way from those faced by operating systems, databases, concurrent languages and any sort of system supporting more than one task executing at the same time.

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 (e.g., network partitioning, crashes, etc.) happening asynchronously with the execution of intentions. It is left to the application developer to provide adequate failure detection and recovery mechanisms.

In addition to the previous limitations, when one considers the requirements arising from the scenarios described in section 3, it can 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 each other. To justify this statement, it has to be remembered that a BDI agent can pursue multiple intentions at the same time. Those intentions have a complex relationship with the agent’s current beliefs and goals and the historical sequence of events being handled. In time, the state of the world can change in such a way that the agent may deliberate to renounce a specific intention and possibly the originating goal altogether.

Consequently, moving an agent to another host or, analogously, checkpointing to disk and recovering later is not just a matter of taking a snapshot of the agent’s memory and copying it into a clone. In fact, the world changes in the meantime and its new state must be taken into account by the reborn agent. Therefore, it is necessary to define a transferable computational state which describes what the agent is trying to achieve and at which point it has arrived, so that it is possible to reason about the new circumstances.

At first glance, this computational state is nothing more than the union of the informational, motivational and deliberative states (Beliefs, Desires and Intentions), i.e. the mental attitudes described in section 2. However, consider the following questions:

- should all beliefs be moved or should they be reconstructed by the reborn agent by probing the world?

- should all goals be moved? How do we distinguish primary, context independent goals from short term subgoals which are generated in a very specific situation which may change while the agent is being moved?

- should intentions be moved or should they be reconstructed by the reborn agent from its list of goals and its updated beliefs about the world?

Neither the architecture nor the existing languages help in resolving these issues. The situation is even more complex when an agent is member of a team. At any given time, the agent is part of a network of relationships built while pursuing private or social goals. Existing BDI systems do not support the explicit expression of boundaries and states of cooperative work.

Current research about BDI is looking at cognitive issues and, for multi-agent systems, is mostly concerned with the notions of shared beliefs, goals
and commitments and how these are built and communicated [KLR+94]. It does not directly address any of the mentioned computational problems. Indeed, the two issues should be kept separated, since they are part of different levels of abstraction. However, the design of a high level architecture is strictly dependent on the capabilities of the infrastructure used for its implementation.

In conclusion, in order to build robust multi-agent systems, we argue that the BDI architecture should incorporate an appropriate computational model. This model should provide a framework to solve the issues arising from concurrency and exception handling and should clarify the relationships amongst the activities being performed by the agent, its mental attitudes and the work being done in a team. Such a model would allow an analysis of the behaviour of an agent at an appropriate level of abstraction and, consequently, the writing of effective meta-level policies, including those required by mobility and recoverability. All of these computational requirements are addressed by the prototype discussed in the next sections.

5 The BDIM Toolkit

In order to investigate possible solutions to the issues raised in section 4, it is necessary to perform a variety of experiments. For this purpose, we built a very basic toolkit in Java, called the BDIM Toolkit. We then used it to develop a prototype of a computationally robust BDI multi-agent architecture, TOMAS (Transaction Oriented Multi Agent System), discussed in section 6. The design of the BDIM Toolkit and TOMAS are discussed in detail in [BR97].

Java was an almost unavoidable choice. The language offers a number of attractive features: platform-independence, a sufficiently rich standard class library including network programming facilities, support for dynamic loading of classes, and suitability for rapid prototyping. It misses the power of logic and functional languages; however, it is at the right level of abstraction for experiments with architectures. Moreover, the availability of software and tools for Java is increasing, including support for mobile agents.

The toolkit is composed of two main layers (implemented as Java packages). The first layer, BDIM, is an architecture-independent definition of the basic elements of a BDI agent. Its purpose is to define an interface between the application (a specific agent of a team) and the run-time system (the framework supporting its execution). The second layer, SimpleBdi, is a stereotypical implementation of an unsophisticated BDI run-time architecture. It is based on an extremely simple computational model, very similar to those adopted by other BDI systems. Its goals are: to provide the simplest possible run-time support needed by applications; to show the underlying problems with its model; and finally to provide a basic set of reusable kernel
mechanisms for more refined architectures.

5.1 BDIM

The BDIM (Belief, Desire, Intention, Message) package is a framework for the development of an agent working in a team. Figure 2 depicts the model adopted by BDIM. Its components are designed to be extremely simple and independent of a specific architecture. They are classified as mental components, capabilities or execution components.

The mental components of an agent include Beliefs, Goals, Messages, Events and Intentions. All these objects are simple data structures manipulated by the application in cooperation with the run-time system. Their purpose is to control the behaviour of the agent.

Beliefs are modelled as records of a relational database. A Belief may be qualified by the application as volatile. A volatile Belief represents some state of the world which may change without control of the agent; transient information internal to the agent itself could also be volatile.

Messages are data structures exchanged among agents. They contain two sets of name/value pairs, one reserved for the application (values), the other for the run-time system (attributes).

A Goal in BDIM is conceptually analogue to the request/answer messages of a remote procedure call: it contains an identifier of the goal to be
achieved, a set of initial (input) name/value pairs and another set (output) filled in when the goal has been accomplished.

An Event contains a type which identifies what happened (a new Goal has been submitted to the run-time system, a Belief has been asserted or retracted from a database, a Message has arrived) and an argument which is the Goal, Belief or Message causing the event.

Finally, an Intention contains the Event which triggered the Intention itself, a Plan and a Plan Interpreter (both discussed below) which have been instantiated to react to the Event.

The capabilities of an agent are supplied by the application developer. They are sets of components derived from two BDIM abstract classes, Plan and IntentionInstantiator.

A Plan represents a recipe to perform a task. It is an aggregation of variables and steps. Each step can manipulate beliefs, send and receive messages or queue them for later delivery, submit new goals, and interact with a human operator. A step can also execute another Plan as a subplan.

A Plan may contain preconditions, represented by a query on a database, a second query that must be satisfied during its execution (maintenance condition) and two termination steps. One termination step is executed in the case of success and the other in the case of failure.

Notably, a Plan does not specify either a triggering condition or the order of execution of its steps (sequential, parallel, with or without backtracking, etc.). A graph of sequences of steps tells only what the acceptable orders of invocation are for a successful execution.

The IntentionInstantiators link Events, Intentions, Plans and Plan-Interpreters. Their purpose is to create a set of applicable Intentions, each containing one Plan/PlanInterpreter couple. When there is an Event to react to, the run-time system invokes all the relevant IntentionInstantiators created by the application. When invoked, an IntentionInstantiator can query the current Beliefs of the agent and determine how many Intentions to create, if any, and which Plans and PlanInterpreters to instantiate for each Intention.

The execution components (the EventIntentionManager and the Plan-Interpreters) are architecture specific classes, i.e., they are supplied by the run-time system.

The EventIntentionManager (EIM) represents the core of the system: it is notified of Events, invokes the IntentionInstantiators to collect the Intentions which are applicable for a certain Event, selects which Intentions to execute and schedules them. Moreover, the EIM can instantiate one or more types of PlanInterpreters as directed by the application.

The PlanInterpreters call the steps of Plans and provide them with operational interfaces. These interfaces are needed to access the databases of Beliefs, to communicate with other agents, to submit new goals and to perform other actions.
By choosing different types of PlanInterpreters, the application selects the policies of execution of its Plans. These policies may differ in the order of invocation of steps, backtracking support, views on databases, mechanism of message delivery, or other characteristics. The information about the current state of a Plan is maintained by its PlanInterpreter and is called the execution context. The EventIntentionManager supplies default PlanInterpreters suit its own preferred mode of execution. However, specific types of PlanInterpreters can be chosen by the IntentionInstantiators. A specific PlanInterpreter can be requested also when invoking subplans; this allows the combination of different policies in the same Intention.

BDIM has no provision for meta-level planning, since all the decisions related to meta planning are encapsulated in the execution components. In other words, meta-planning is performed in BDIM when a specific EventIntentionManager is instantiated at startup and whenever a specific type of PlanInterpreter is selected for the execution of a Plan.

5.2 SimpleBdi

SimpleBdi is a simplified but parallel implementation of [RG92], which mimics components and behaviour of [GI89]. Figure 3 depicts how its components cooperate.

SimpleBdi contains a single database of Beliefs, shared by all running
Intentions. Its GUI is very rudimentary but supports any number of dialogues with the human operator as required by the Intentions being executed concurrently. Team communication is supported by an unreliable, message-oriented transmission channel.\(^1\)

Only one type of PlanInterpreter, named **BdiPlanInterpreter**, is supported. It executes the steps of a Plan one at the time, following one of the sequences specified in the Plan graph. If one step fails, backtracking is limited to the last successful step and an alternative sequence is attempted. It continues until either a sequence is completed (success case) or no alternatives can be found (failure). The Plan is not started if its pre-condition is not satisfied. Failure is also forced if the maintenance condition is no longer satisfied prior to the termination of a sequence of steps or if immediate termination is required by the EventIntentionManager.

BdiPlanInterpreter serializes all the accesses to the database by taking per-process locks during the execution of steps manipulating Beliefs.

Events are generated by BdiPlanInterpreter whenever the executed Plan performs operations on Beliefs or submits Goals. Events are also generated by the communication subsystem when a Message is received. Finally, Events can be generated by any Java object not part of the agent itself in order to submit Goals (for instance, to provide the initial set of goals at startup).

The SimpleBdi EventIntentionManager starts a new Java thread for each Event it is notified of. This thread invokes the relevant IntentionInstantiators, collects the applicable Intentions and tries to run one of them, that is, it requests the execution of its first Plan to its PlanInterpreter. Similar to [G189], these actions are repeated until either a Plan terminates successfully or no applicable intention is generated which is different from those which have been already tried.

The EventIntentionManager can be required to interrupt the service of an Event at any time. For instance, it may be alerted by the application when a certain Goal should not be pursued any more. The interruption forces the failure of the Plan of the current Intention serving the Event and all its subplans, if any.

SimpleBdi is plagued by all the problems discussed in section 4. However, it provides a good foundation for different architectures, such as TOMAS discussed in the following section.

\(^1\)For both SimpleBdi and TOMAS, a team is just a set of agents which can communicate with each other, i.e., that know their communication end-points. Any organizational or functional structuring is left to the application.
6 TOMAS

TOMAS (Transaction Oriented Multi Agent System) is an extension of SimpleBDi, whose components are all reused. TOMAS has the additional ability to interpret a hierarchy of intentions pursued in parallel by a team of agents and related by a causal chain of goals and messages as a distributed nested transaction.

6.1 Transactions

Transaction systems are very common and widely used; they include databases, transaction operating systems, certain concurrent languages, etc. A transaction, as it is commonly defined [GR93], is an ACID unit of work (Atomic, Consistent, Isolated, and Durable).

Atomicity means

all-or-nothing: either all actions happen or none happen.

It is the responsibility of the transaction system to guarantee the atomicity of transactions.

Consistency, the correctness of the transformations performed by a transaction, is mostly a problem for the developer when writing transactions. However, a transaction system may automatically enforce some constraints (e.g., invariants on relations in a database) and force the abort of a transaction if they are violated.

Isolation is the property that two transactions running in parallel have the illusion that there is no interference, as if the system were running one transaction at the time. Isolation is normally obtained by using a locking scheme to prevent two transactions from updating the same data in a conflicting manner.

A transaction manager tracks which resources (that is, objects) are affected by a transaction. When a transaction eventually terminates, either successfully or with failure (i.e., it commits or aborts), the transaction manager informs and coordinates the resource managers. Often, this coordination is obtained by adopting a well known protocol, Two Phase Commit, which guarantees that either all the resource managers make the modifications durable or none do.

The work done by a transaction can affect many resource managers distributed on various processes and hosts of a network. A transaction itself is called distributed when it involves multiple applications and multiple transaction managers.

Systems supporting nested transactions, either distributed or not, are less common than those supporting normal (flat) transactions. A nested transactions is composed by a root (or top level) transaction and a number of subtransactions. A subtransaction is a
transaction that is nested under a higher-level transaction. Of the ACID properties, it has atomicity, consistency and isolation but is not durable. Even after a subtransaction has locally committed, it can be rolled back due to the abort of the parent transaction.

A hierarchy formed by a toplevel transaction and all its descendents is called family.

The advantage of having subtransactions is that, if one fails, it aborts all the actions performed by itself and its descendents but neither its siblings nor its parents are affected. In other words, nested transactions provide a framework for managing partial failures.

6.2 Plans as Nested Transactions

In addition to BdiPlanInterpreter taken from SimpleBdi, TOMAS supports a family of PlanInterpreters called TranPlanInterpreter. A TranPlanInterpreter executes its Plan as an ACID transaction, either root or subtransaction of a parent. For brevity, a Plan being interpreted by a TranPlanInterpreter will be referred to as a transactional plan.

Most of the operational interfaces offered to the steps of a transactional Plan manipulate transactional resources. Locks are automatically taken in order to guarantee the isolation of the transaction. Different kinds of TranPlanInterpreters take different types of lock; both concurrent and exclusive
access to resources are supported. The lock manager supports transaction nesting: if a lock $L$ is granted to a transation $T$, then $L$ is implicitly granted to all the subtransactions of $T$. When $T$ commits, its locks are inherited by its parent or released if $T$ was root. Analogously, appropriate care is taken by all the transactional resources in order to guarantee a **consistent** view of the information being manipulated.

The effects of a nested transaction are *committed*, i.e. become a permanent part of the state of the agent, only if and when the Plan corresponding to the root of the family terminates successfully. The failure of a transactional Plan or one of its transactional resources forces the failure of the transaction and all its descendent (possibly the whole tree if root), and consequently no modification is committed.

Any Plan can become transactional, either explicitly (that is, when required by an IntentionInstantiator or by a Plan invoking a subplan) or implicitly by the defaults applied by TOMAS. The transaction of a Plan is created before checking its preconditions. By default, this transaction is root unless the Plan has been created as a subplan or to serve a Goal or Message generated by another transactional plan, in which cases it becomes subtransaction of the latter. Figure 4 shows an example of nested transaction in a single agent.

The application can control the extent of a nested transaction. Transactional and non-transactional plans can be freely mixed even during the execution of a single Intention. However, the application programmer must take sufficient care when making such a choice.

TOMAS has two main transactional resources, its **belief database** and a **message queue**. Other resources (for instance, relational and object oriented databases, store and forward message queues, printing and electronic mail systems, any other application-specific resource internal or external to an agent) can be easily added.

The belief database is a repository which allows multiple views on **non-volatile** beliefs; these views are designed for many uses such as transactions and **what-if reasoning**. The **main store** is the same relational database of SimpleDb. However, the **view** manipulated by a transactional Plan consists of by the main store modified by all the assertions and retractions performed by the Plan itself and all its parents. Different branches of the same family can have different views. The view of a subtransaction is merged with the view of its parent when the subtransaction commits. The eventual commitment of the root of the nested family merges all the modifications with the main store.

Events related to the assertion and retraction of non-volatile beliefs are generated only if the root of a nested transaction commits.

Note that volatile beliefs are not under transactional control. Also, non-transactional plans directly manipulate the main store without any serialization; appropriate care is taken to guarantee the consistency of view to
Figure 5: A simple case of a distributed nested transaction

the transactional plans, even in those cases.

The Messages queued by transactional Plans are stored in a transient repository and sent when the root transaction commits or discarded otherwise. The next section discusses the management of the informational and directional Messages exchanged among agents during a transaction.

6.3 Cooperative Work as Distributed Nested Transactions

The communication subsystem of TOMAS supports reliable, peer-to-peer connections among agents. It cooperates with the distributed transaction manager of TOMAS, of which each agent has its own instance, to support transaction propagation and the standard Two Phase Commit Protocol.

Whenever a request or an informational message is sent by a transactional Plan of an agent, the nested transaction mode is automatically extended to the destination agent. Consequently, the parent of a transaction being performed by an agent can be in a different agent and its subtransactions may be running on other agents of the same team. Figure 5 depicts a simple example of a distributed nested transaction in a team of three agents but “re-entering” the agent which initiated the work.

The service of a Message by its destination is considered a transactional
action of the Plan which sent the Message. The implication is that the receiving agent must successfully react to the Message, that is, there must be one Intention successfully terminating (possibly after previous Intentions had failed). Any Message that the receiver is not able to handle forces the failure of the transaction which sent it. The default PlanInterpreter for an Intention serving a transaction-propagating Message is the same TranPlanInterpreter of the sender Plan.

A distributed transaction family commits in all the involved agents when the root on the initial agent commits. If an agent crashes, all the affected branches of the transactions it was involved in will automatically fail in all the surviving agents; in other words, the crash of an agent is just a case of the partial failure of a nested transaction.

6.4 Current State and Limitations

TOMAS is currently under testing and the first experiments are being attempted.

An extremely simple set of banking examples, with one agent acting as a branch and another agent as the central site, has been developed. When SimpleBdi is adopted as their run-time system, the agents are plagued by inconsistencies of their Beliefs due to race conditions. Since the examples do not manage failure, even trivial errors (for instance, trying to withdraw more money than what is available in one’s account) may cause the agents to behave incorrectly or unexpectedly.

By simply selecting TOMAS as their run-time system, the same examples automatically perform the correct serialization of accesses to the Beliefs. They behave in a correct and reliable way even when errors are detected by Plans or an agent crashes. By appropriately organizing Plans and IntentionInstantiators, partial failures are automatically managed by the Event service mechanism described in section 5.2.

The burden imposed by multi-agent Two Phase Commits of each sub-transaction of a distributed transaction can be substantial. It must be noted, however, that there are algorithms (not implemented by TOMAS at this stage) that automatically reduce the number of messages being exchanged or even avoid the Two Phase Commit in certain circumstances [GR93].

ACID transactions are not appropriate for long term activities. It is a common industrial practice, although, to break those activities into sequences of short term transactions, without adopting any esoteric transaction model. In the case of TOMAS, a long term Intention could have a non-transactional Plan invoking sequences of transactional subplans or submitting Goals served by transactional plans. Compensatory transactions may be required if failure happens outside of the control of a transaction.
7 Computational State in TOMAS

Far from resolving all the issues described in section 4, TOMAS, however, starts to tackle the problems at their core.

Transactions move the state of the Beliefs of a team of agents from one consistent state to another. The ACID properties make the analysis of these state transitions extremely easy [GR93], without imposing any arbitrary sequentialization of the execution of the steps of concurrent Intentions. If failures occur during the lifetime of a nested transaction, the transaction managers automatically restore a consistent state in all the involved agents. Partial failures are addressed by the combination of the nested transaction model with the Event service mechanism described in section 5.2; this makes fairly easy to build agents which are resilient to many categories of faults.

A distributed computation performed as a transaction by a team of agents has clear boundaries in time and space. We have not sacrificed the flexibility and reactivity of the BDI model in TOMAS. Furthermore, such flexibility is now available to the transaction processing world, while a clear and well known set of engineering techniques can be adopted when defining the cooperative work of teams of agents.

Some smaller but relevant work has also been done on the mental components of an agent. The classification of Beliefs as volatile or non-volatile gives at least a partial answer to the questions in section 4. While volatile Beliefs can always be reconstructed by probing the world, non-volatile beliefs are primitive information representing historical memory accumulated during the activity performed by the agent; as such, they are intimate part of the identity of the agent and must be preserved when moving or restoring it.

Goals and Messages propagating a nested transaction can now be classified as secondary, while the Events activating a root transaction are clearly "more important". It becomes feasible to design a transferable, possibly permanent store for the Events which are not generated as part of a transaction; these events will be called root Events.

A TOMAS agent could be moved to a new location by adopting the following algorithm. It is executed, for instance, on receiving an appropriate Event such as one carrying a "move agent" Goal:

- checkpoint the state of the agent and force the immediate failure of all the Intentions currently being executed;

  This causes any active transaction to be rolled back, leaving a consistent state of the Beliefs in all the affected agents.

- transfer all the root Events which were being served to the new location of the agent;
• transfer all the non-volatile Beliefs;

• on the reborn agent, invoke an application startup routine which re-
stores the state of volatile Beliefs;

• finally, let the EventIntentionManager restart the service of the trans-
ferred root Events.

In other words, the agent redefines its Intentions from the
original Events but in the new context of the world.

In practice, the above algorithm is not inefficient as we expect that not too
many subtransactions will need to be rolled back when it is applied. This is
because:

• the “move agent” event is a programmed event fired only when an
agent is in certain states (for example, all the current transactions
have been completed);

• the agent may choose to delay the movement until it is convenient (for
instance, the cost of moving is low).

In any case the above algorithm can be safely applied independently of the
state of the agent.

Among other objectives, further research on TOMAS should investigate
PlanInterpreters which are able to checkpoint the execution context of their
Plans and recover it after a forced suspension. These PlanInterpreters may or
may not support transactions. If their execution context were transferable, it
would become possible to move certain Intentions in the algorithm outlined
above.

8 Comparisons

TOMAS has strong similarities to concurrent languages and environments
supporting fault tolerant computation and the mobility of objects (for in-
stance, Arjuna [PSWL95], DOME [BSS94], Electra [Maf95]).

An interesting and very recent stream of research about multi-agent
systems, inspired by advanced transaction concepts, is Interaction Oriented
Programming (IOP) [Sin96] [Sin97]. IOP is essentially concerned with the
coordination of agents which cooperate to achieve a common task, possibly
over a long period of time. Its aim is to provide a reliable and flexible
framework for teams, while not being directly involved with the internal
computations performed by each single agent.
9 Conclusions and Future Work

Intelligent agents are an extremely lively area of research and application of Artificial Intelligence. They could take advantage of mobility services as much if not more than any other complex distributed system.

BDI is one of the most promising architecture for intelligent agents and significant research has been done regarding teams of cooperating BDI agents. In their current form, BDI agents cannot be mobile. We have identified the problems with BDI in its computational model. We have proposed an alternative architecture, TOMAS, that makes agents more robust, easier to analyse and more predictable than the BDI model.

While TOMAS does not directly address mobility, it is suited to support it; an algorithm for moving an agent has been proposed. The next step of research regarding mobility in TOMAS should define a transferable store of Events and examine adding new models of execution of Plans.

The transaction model at the core of TOMAS is very well known and of general applicability. We think that the research illustrated here is of more general interest than BDI agents; the issues being tackled and the solutions proposed should be reusable in many types of AI and industrial applications.

Acknowledgements

The authors would like to thank Evan Harris for his precious comments.

The research was supported by an Australian Research Council grant, the Key Centre for Knowledge Based Systems, and the Cooperative Research Centre for Intelligent Decision Systems.

P. Busetta was supported by an Australian Postgraduate Award (APA) scholarship.

References


