Active Databases and Agent Systems - A Comparison

James Bailey[1]
Michael Georgeff[2]
David Kemp[1]
David Kinny[1]
Kotagiri Ramamohanarao[1]

Technical Report 95/10[3]

Department of Computer Science
University of Melbourne
Parkville 3052

Abstract

This paper examines Active Databases and Agent Systems, comparing their purpose, structure, functionality, and implementation. Our presentation is aimed primarily at an audience familiar with active database technology. We show that they draw upon very similar paradigms in their quest to supply reactivity. This presents opportunities for migration of techniques and formalisms between the two fields.

---

[1]{jbailey,kemp,dnk,rao}@cs.mu.oz.au

[2]georgeff@aaii.oz.au

[3]Appears in T.Sellis, editor, *Proceedings of the Second International Workshop on Rules in Database Systems*, Lecture Notes in Computer Science 985, pages 342-356, Athens, Greece, 1995.

1 Introduction

In recent times, two technologies have become prominent in the database and artificial intelligence research communities. An Active Database (ADB) is a system which supplements traditional database functionality by reacting automatically to state changes, both internal and external, without user intervention. An Agent System (AS) is a software system which utilizes agents that possess certain mental attitudes which allow them to emulate "intelligent" behaviour.

Although these technologies have been developed in different fields of computer science and are designed with different applications in mind, the methods they employ to achieve their ends have much in common. It is the fact that they address the problem of generating reactive systems which draws these two seemingly disparate models together. Indeed, it is our contention that researchers in both of these areas would benefit from exposure to ideas from the other one, particularly as research in the respective communities has tended to focus on different problems.

Agent systems are usually employed in applications where complex, purpose-directed behaviour is needed in a changing environment. There are many different areas that require these abilities; air traffic control, control and monitoring of industrial processes, network management and management of business processes including workflow and automated trading, to name a few. Applications of active databases are also diverse. They can range from simple database alerters to systems that perform complex database changes.

Interestingly, initial applications of ADBs have been relatively unambitious and have concentrated on issues such as extending the handling of integrity constraints [7] and providing view maintenance [8, 20]; areas also of importance to passive databases. This reflects the inherent difficulty of reasoning about systems that react in a complicated way and perform a series of complex tasks, which is also the reason for the limited functionality of current commercial systems. Once some of these tractability problems have been overcome, however, the likelihood of advanced applications of ADBs being successfully implemented should increase. The incorporation of AI techniques may become feasible, which will result in ADB researchers confronting some of the issues that AS researchers presently face.

In this paper we identify some of the key similarities and differences between ADBs and ASs. We focus mainly upon issues in ASs of interest to the ADB community, presenting in some detail one of the more mature AS paradigms. This is not to suggest that ASs are homogeneous — there is much diversity amongst them. In fact, with the recent application by commercial vendors of the term "agent" or "software agent" to simple software offerings of limited functionality, the term is in danger of becoming just another marketing buzzword.

We commence by describing the key features of the selected agent architecture. Next, we contrast this with typical ADB architectures and identify areas of commonality. Here our emphasis is on a few key comparisons rather than an exhaustive list of similarities and differences. Familiarity with ADBs is assumed[1]. We present examples to illustrate both the expressiveness of the languages typically provided and the classes of problems where a particular language may be more suitable. After making comparisons in several other areas, we look at research trends in each field and propose topics for future research.

[1] Detailed descriptions of ADBs may be found elsewhere [12, 3].

2 Agent System Architectures

The term *agent* has been used in AI to label a multitude of systems, ranging from simple reactive systems devoid of explicit symbolic representations [29, 1, 14, 26] through to complex heterogeneous multi-agent systems with individual and shared representations [21, 10, 13]. The term has recently achieved the status of "flavour of the month", with many commercial software developers offering or promising "software agents" that will act intelligently and autonomously to perform useful tasks. Unfortunately, as with many previous developments in AI, there is a danger that a technology that is still far from mature will be oversold, leading to a negative reaction from the marketplace.

Despite this terminological confusion, there has been considerable progress made in research in AI into the design of agents, by which we mean reactive, knowledge-based systems that are embedded in some dynamic, partially unpredictable environment. The essential requirement for such agents is that they should be able to take actions to achieve certain goals, while responding in a timely and considered manner to changes in the environment. This requires amongst other things, an ability to determine appropriate actions in a context-sensitive manner and to deal with failure and uncertainty.

The goals of the agent, intrinsic or supplied by a user, may also change over time. Thus an agent must also be able to reason about which goals to pursue, how best to achieve them given the limited resources it has available, and what to do in the case of goal conflict.

In this paper we will focus on a particular approach to software agency that has achieved considerable maturity, the *Belief/Desire/Intention* (**BDI**) architecture [16, 15, 11, 27, 6, 31, 25, 28]. This architecture has its origins in the study of *mental attitudes*; namely, Beliefs, Desires and Intentions that represent, respectively, the agent's information, motivational, and deliberative state. These mental attitudes directly determine the system's behaviour and are critical for achieving adequate performance when deliberation is subject to resource bounds [5, 23].

2.1 BDI Agent Architecture

Figure 1 shows the essential structure of a typical BDI agent[2]. It consists of a *database* containing current *beliefs* or facts about the world; a set of current *desires* (or *goals*) to be realized; a set of *plans* describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations; and an *intention structure* containing those plans that have been chosen for [eventual] execution. An *interpreter* manipulates these components, selecting appropriate plans based on the system's beliefs and goals, placing those selected on the intention structure, and executing them.

The system interacts with its world through its database, which acquires new beliefs in response to changes in the world detected by its sensors, and through the actions that it performs via its effectors as it executes its intentions. A user of the system may directly query and update the agent's belief database.

The interpreter runs the entire system. From a conceptual standpoint, it operates in a relatively simple way. At any particular time, certain goals are established and certain events occur that alter the beliefs held in the system database. These changes in the system's goals and beliefs invoke (trigger) various plans. One or more of these applicable plans will then be chosen and placed on the intention structure. Finally, the interpreter selects an executable

[2]The architecture described is based upon PRS [17].

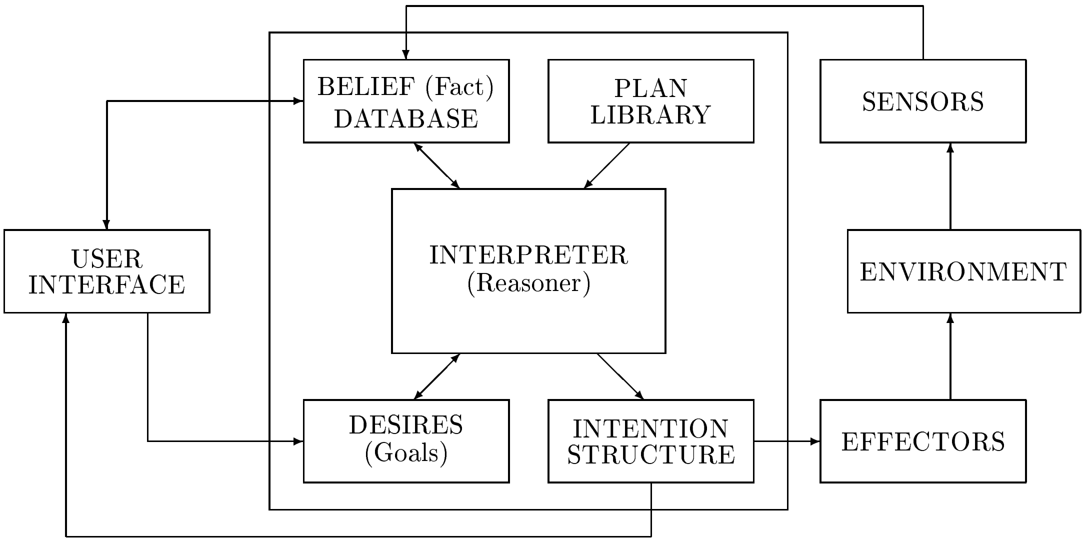

Figure 1: BDI Agent Architecture

intention from the intention structure and executes *one step* of that intention. This will result in either the performance of an action, the establishment of a new subgoal, or the conclusion of some new belief. At this point the interpreter cycle begins again: the newly established goals and beliefs trigger new plans, one or more of these are selected and placed on the intention structure, and again an intention is selected from that structure and partially executed.

2.1.1 Beliefs

An agent's belief database contains its current knowledge about the state of the world and also, possibly, some aspects of its own internal state. Typically, this will include facts about static properties of the application domain, such as the structure of some subsystems or the physical laws that must be obeyed by certain mechanical components. Other beliefs are acquired by the agent itself as it executes its plans. These will typically be current observations about the world or conclusions derived by the system from these observations, and these may change over time. In a multiagent system, an agent may also need to maintain various beliefs about other agents, including beliefs about *their* mental states.

The knowledge contained in the database is often represented as ground instances in first order predicate calculus. For example, a belief about the pressure in a fuel tank might be represented:

```
pressure(fuelTank,300)
```

Beliefs about internal system states are called *metalevel beliefs*. These typically describe the beliefs, goals, and intentions of the system, as well as other important control information. For example, a metalevel belief that a particular intention is suspended may be represented:

```
intentionStatus(intention235,suspended)
```

2.1.2 Goals

Goals are expressed as conditions over some interval of time (i.e., over some sequence of world states) and are described by applying various temporal operators to state descriptions. This allows the representation of a wide variety of goals, including goals to achieve, test, maintain and wait for given conditions. For example, the goals to test whether the pressure in a tank is less than 200 psi, to wait for the pressure to become greater than 285 psi, and to achieve (bring about) that the pressure is 350 psi may be represented as follows:

`test pressure(fuelTank,X) and X` $<$ `200`

`waitfor pressure(fuelTank,X) and X` $>$ `285`

`achieve pressure(fuelTank,350)`

A given action (or sequence of actions) is said to *succeed* in achieving a given goal if its execution results in a behavior that satisfies the goal description. Goals of achievement may succeed, vacuously, if the agent already believes that the specified state holds.

Goal descriptions are not restricted to specifying desired behaviors of the external environment but can also characterize the internal behavior of the system. Such descriptions are called metalevel goal specifications.

2.1.3 Plans

An agent's plans are the elements of its knowledge base that describe how the agent should react when certain facts are added to its belief database, or when it newly acquires certain goals. These new facts may have arisen due to events in the external world being directly *perceived*, or by the fact being *inferred* by the agent itself; that is, by the execution of some intention. New goals, similarly, may arise as top-level (intrinsic) goals, supplied to the agent by a user of the system (or, perhaps, by another agent), or as subgoals of an existing intention.

Each plan consists of an *invocation condition*, which specifies upon which events the plan should be triggered, a *context condition*, which specifies under what situations the plan applies, and a *body*, which describes the steps of the procedure. Together, the invocation condition, context condition, and body of a plan express a declarative fact about the results and utility of performing certain sequences of actions under certain conditions.

The invocation condition is an event expression that describes what must occur for the plan to be considered for execution. Usually, this will be the acquisition of some new goal (in which case the plan is invoked in a goal-directed fashion) or some change in system beliefs (resulting in data-directed or reactive invocation).

The context condition is a state expression that describes the belief state that must hold for the plan to be executed. Often, this expression will contain free variables (which also occur in the body) that will be bound by its evaluation, thus what is intended is a particular *instance* of the plan[3].

The body of a plan is represented as a graph with one distinguished start node and possibly multiple end nodes. The arcs in the graph are labeled with the actions to be performed or subgoals to be achieved in carrying out the plan. Successful execution of a plan consists of performing or achieving each of the actions or subgoals labeling a path from the start node to

[3]When the context condition can be satisfied in more than one way, the interpreter must choose which instance of the plan to intend.

an end node. This formalism allows richer control constructs (including conditional selection, iteration, and recursion) than most other plan representations.

The set of plans in a typical agent consists not only of procedural knowledge about a specific domain, but also includes *metalevel* plans — that is, knowledge about the manipulation of the beliefs, desires, and intentions of the agent itself. Metalevel plans allow the agent designer to override the default behaviour of the agent's interpreter. For example, typical metalevel plans encode various methods for choosing among multiple applicable plans, modifying and manipulating intentions, and computing the amount of reasoning that can be undertaken, given the real-time constraints of the problem domain.

Plans may also have another condition associated with them called a *maintenance condition*. Like the context condition, it must be satisfied for the plan to be invoked, but it must then remain true throughout the execution of the intention. If at any time it becomes false then the intention is terminated. Maintenance conditions provide a mechanism for preventing interference between parallel intentions, as well as guarding against change in the world that renders a particular plan no longer appropriate under certain circumstances.

2.1.4 The Intention Structure

The intention structure contains all those tasks, called *intentions*, that the system has chosen for execution, either immediately or at some later time. A single intention consists of a top-level plan instance together with all the (sub-)plans that are being used in attempting to perform that plan. It is directly analogous to a *process* in a conventional programming system.

At any given moment, the intention structure may contain a number of such intentions, some of which may be suspended or deferred, some of which may be waiting for certain conditions to hold prior to activation, and some of which may be metalevel intentions for deciding among various alternative courses of action.

For example, in handling a malfunction in the propulsion system of the NASA space shuttle, the system might have, at some instant, three intentions in the intention structure: one suspended while waiting for, say, the fuel-tank pressure to decrease below some designated threshold; another suspended after having just posted some goal that is to be accomplished (such as interconnecting one fuel subsystem with another); and the third, a metalevel procedure, being executed to decide which way to accomplish that goal.

The set of intentions comprising the intention structure form a partial ordering with possibly multiple least elements (called the *roots* of the structure). An intention earlier in the ordering must be either realized or dropped (and thus disappear from the intention structure) before intentions appearing later in the ordering can be executed. This precedence relationship between intentions enables the system to establish priorities and other relationships between intentions.

The intention structure captures, conceptually, the decisions an agent has made about the means to achieve its goals. Because an intention is a process, sometimes long-lived, the agent is, in a real sense, *committed* to achieve these goals in that manner. When actions fail, or the world changes in unexpected ways, the agent may need to *reconsider* these commitments. The right balance between commitment and reconsideration is an important agent design issue which depends sensitively on the properties of the world in which the agent exists [23].

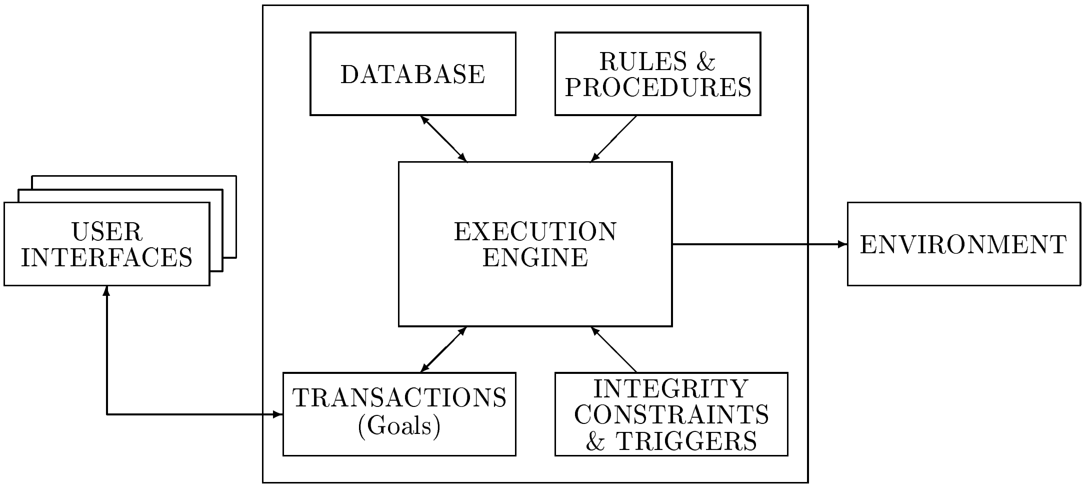

Figure 2: Active Database Architecture

3 Comparison of Architectural Features

The database component in an ADB is analogous to the belief database in an AS. The contents of the database represent the facts of the world that the system is modelling — these correspond to the agent's beliefs. Example facts are a robot's position and battery charge level, or the balance of a given account and the number of withdrawals made. Changes are made to the database by transactions to reflect the changing state of the world.

ADBs permit the enforcement of consistency on parts of the database by integrity constraints and/or triggers. ASs typically offer only limited forms of integrity constraint, but maintenance conditions may be used to terminate intentions if consistency assumptions are violated, and fact-invoked plans may be employed to maintain and recover consistency.

In an ADB, users can typically ask arbitrary queries about the contents of the database and the DBMS acts as a broker that delivers this information. The belief database in an AS can be viewed as having two "users" - the agent itself and the outside world (which may include humans and other agents).

The attainment of goals in an ADB can be achieved by rules and update procedures. In an AS, plans are the means used to achieve goals. Compared to the action languages used in ADBs, AS plan languages typically provide more powerful operators and a rich set of constructs for combining actions and subgoals.

Events in both systems are similar, with their semantics varying between implementations. Typically the events in an ADB are restricted to database operations, but this need not be the case. Some ADBs provide a rich language for combining events, but ASs tend to restrict themselves to atomic event occurrences. They do, however, possess the ability to generate a variety of metalevel events in the course of plan execution. These may be useful for allowing plans to reason about one another, an activity which is not natural for an ADB to mimic.

The occurrence of events in an ADB causes the activation of triggers which in turn evaluate conditions and possibly execute actions. Events in an AS cause the invocation of plans which are executed after placement on the intention structure.

ADBs are designed with multiple user access in mind. Users specify their needs by means of transactions and submit them to the database for execution. Conceptually, these transactions

	Active Database	Agent System	
	Events	Events	
Rules	Event Predicate	Invocation Condition	Plans
	Condition	Context Condition	
	Action	Plan Body	
	User Transactions	Intrinsic Goals	
	Database Facts	Beliefs	
	Update Procedures	Goal-invoked plans	
	Integrity Constraints and Triggers	Event- or Fact-invoked plans	

Table 1: Similar Concepts in Active Databases and Agent Systems

represent the top-level goals given to the database, and the statements contained within a transaction give rise to various subgoals. Transactions may cause alteration of database facts, output to the world, and change in the rule structure. They will typically compete for resources and, if the database is dedicated to a particular application, may have a fixed structure. The intrinsic goals in an AS will commonly be defined by the designer at the agent's conception, but may also be established at runtime by the execution of the agent's plans, a user, or other agents.

In an ADB, ACID properties [18] are an inherent part of the behaviour of transactions, whereas in an AS this functionality has to be explicitly considered in the design of plans. Multiple updates (or other actions) may be performed as a single, atomic step, but more complex sequences must employ explicit semaphores or priority-based methods to guarantee exclusion.

Atomicity is harder to achieve, however, since no notion equivalent to transaction rollback (due to violation of integrity constraints) exists in ASs. This is because the actions performed will typically affect the external environment, and to undo these would require an ability to "roll back the real world". Concurrency is not usually a problem because agents do not share their belief database with other agents. Where agents need to update each other's beliefs (or goals), it can be accomplished by message passing. When multiple goals need to be performed concurrently, it is the responsibility of the agent designer herself to structure their execution so that data consistency is maintained.

Execution strategies are similar in both systems, being controlled by a high-level manager/interpreter. In an ADB this runs submitted transactions by drawing upon information in the database, rule and trigger repositories (see Figure 2). In an AS plans are invoked not only by goals but also by database changes and other events, resulting in a mixed top-down/bottom-up strategy.

Differences in execution strategy also exist with respect to how multiple simultaneous events and multiple triggered rules/plans are handled. For example, in an ADB, one typically wants to execute all rules that are triggered by a particular event, whereas in an AS one typically wants to invoke only one plan in order to achieve a goal — only if that plan fails will a different one be adopted.

The plans executed by an agent may be long running processes and possibly never terminate. ADB (trans)actions, conversely, are typically simple operations whose lifespan is limited. ASs have a notion of success/failure associated with actions, plans, and goals. This is typically not supported in active databases, and limits the extent to which they can interact with the outside world, where it may be necessary to make several repeated attempts to achieve a desired outcome [19].

4 Example: MicroBlue versus BigSoft

Consider a system that tracks the stock market and is capable of sending buy and sell orders to a stock broker. Here we compare how this system might be implemented in an ADB and in an AS. One should note that the example we present is for illustration purposes only and not necessarily indicative of the way things would be implemented in a real application. The interface to the broker consists of two types of messages that can be sent to the broker and one type of message that can be received from the broker. The system can send a *trade* message (`trade`(*buyOrSell, name, amount*)) to request that the broker buy or sell a number of shares in a company. For example, `trade(sell,‘Blue’,40000)` is a request to sell 40,000 shares in the company called ‘Blue’.

If the broker is unable to carry out a buy/sell order — because, for example, no one is buying a stock that is offered for sale — then the broker sends an *orderFailed* message (`orderFailed`(*name*)) to the system. For example, `orderFailed(‘Blue’)` would be received if the broker could not sell the 40,000 shares that she was requested to sell (we are assuming that we do not request the broker to both buy and sell the same shares at the same time).

The system can also send a *terminate* message (`terminate`(*name*)) to request that the broker stop trying to buy or sell particular shares. For example, `terminate(‘Blue’)` is a request to stop buying or selling ‘Blue’ shares.

To give examples of how to implement this system using an ADB or an AS, we should carefully describe the syntax and semantics of the example languages that we use. Instead, we will employ here simple, intuitive languages whose semantics should be reasonably obvious from the examples.

We use the following syntax for specifying a trigger:

`on` *event* `[if` *condition*`] then` *action*

If the trigger has no condition then it may be omitted.

As well as simple insert, delete and update events on relations, we allow received messages to cause events, and events to be combined into compound events. Compound conditions may be created using `and`, `or` and `not`. Actions may be database operations (e.g. insert, delete, update), sending external messages, user defined *signalled events*, and sequences of actions (separated by `andthen`). Signalled events do not increase the expressive power of the language, but merely make its use more convenient.

As we will not be using the full expressive power offered by existing languages, we define our own so as to make the examples more concise than could otherwise be achieved.

4.1 An Active Database Language

In our example language, we use the following two syntaxes for specifying a trigger:

`on` *event* `if` *condition* `then` *action*

If the trigger has no condition then we use the shortened form of:

`on` *event* `then` *action*

Here is a very simple trigger:

```
on insert p(A,B) if A > B then insert q(A,B)
```

The event is `insert p(A,B)`, the condition is `A > B`, and the action is `insert q(A,B)`. We are using a DATALOG notation to refer to tuples in relations. The meaning of the trigger is: if a tuple is inserted into relation `p`, then assign its first attribute to the variable A, and its second attribute to the variable B, and if $A > B$, then insert into the relation `q` the tuple $q(A, B)$.

As well as inserts and deletes, we allow rules to triggered on update events, where the condition and action parts of the trigger must indicate if they are referring to the old or new versions of attributes of the tuple updated. For example:

on update p(A,B) if $A_{old} > A_{new}$ then insert q(A_{new}, B_{new})

This trigger is triggered if a tuple in the relation `p` is updated, and the old value of its first attribute is greater than the new value of its first attribute.

We allow constants to appear in the event part of the trigger. For example:

on update p(A,'Blue') if $A_{old} > A_{new}$ then insert q(A_{new},'Soft')

The semantics of this is that if a tuple in the relation `p` whose second argument is 'Blue' has its first argument (A) updated, and the old value of A is greater than the new value of A, then a tuple is inserted into `q`. In practice, the test that the second argument is equal to 'Blue' would often be regarded as part of the condition rather than being regarded as part of the event, and it could clearly be implemented this way.

The condition can be an arbitrary DATALOG expression. For example, to define a trigger that is triggered on the update of a tuple in `p` whose second argument has the value 'Blue', but whose condition requires that there is currently a tuple in the relation `q` whose second argument has the value 'Soft' and the first attribute of the tuple from `q` is less than the first attribute of the new version of the updated version of the tuple in `p` we would use something like:

on update p(A,'Blue') if q(B,'Soft') and $A_{new} > B$ then insert q(A_{new},'Soft')

We use compound events such that if $e1$, $e2$, and $e3$ are events, then the following are also events:

- $e1$ `and` $e2$: This is true when both $e1$ and $e2$ have occurred in any order with any amount of intervening time and any number of intervening events.
- $e1; e2$: This is true when $e1$ is followed by $e2$ with any amount of intervening time and any number of intervening events.
- $e1$ `or` $e2$: True when either $e1$ or $e2$ occur.
- $e1$ `not in [`$e2, e3$`]` : True if $e2$ is followed by $e3$, but $e1$ does not occur in between.
- $e1$ `not in [`$e2, e2 + n$ `seconds]` : True if $e2$ occurs, and $e1$ does not occur within n seconds of that particular $e2$ occurring.

Note that expressions of the form $e + n$ `seconds` are only allowed to occur in events of the form $e1$ `not in` [$e2$, $e2 + n$ `seconds`]. We do this to ensure that the '$e2$' event occurring in $e2$ and $e + n$ `seconds` are the same event.

We allow users to define names for compound events. For example, the following two specifications are equivalent:

Version 1:
```
on insert p(A,'Blue') or insert p(A,'Soft') then insert q(A)
```

Version 2:
```
e(A) = insert p(A,'Blue') or insert p(A,'Soft')
on e(A) then insert q(A)
```

We allow actions to be database operations (e.g. insert, delete, update); external message posting (e.g. sending a trade message); or a user defined *signalled event*. User defined signalled events do not increase the power of the language, but they do make it more convenient to use. The following two specifications are equivalent. Note that `e(A,B,C)` is a signalled event in version 2:

Version 1:
```
on insert p(A,B,C) if A < B then insert q(A,B)
on insert p(A,B,C) if A < B and B < C then insert r(A,B,C)
```

Version 2:
```
on insert p(A,B,C) if A < B then signal e(A,B,C)
on e(A,B,C) then insert q(A,B)
on e(A,B,C) if B < C then insert r(A,B,C)
```

4.2 Buying Blue Shares

Suppose we wish the system to buy 40,000 MicroBlue ('Blue') shares if the 'Blue' share price goes from less than or equal to the BigSoft ('Soft') share price to greater than this price and stays that way for at least 5 hours.

4.2.1 An Active Database Solution

To solve this in an ADB, we can define two events $e1$ and $e2$ which are signalled by triggers when the 'Blue' share price has gone from not greater than to greater than (respectively, greater than to not greater than) the 'Soft' share price. Here are triggers that could be used:

on update price('Blue', B) if price('Soft', S) and $B_{old} \leq S$ and $B_{new} > S$ then signal(e1)

on update price('Soft', S) if price('Blue', B) and $B \leq S_{old}$ and $B > S_{new}$ then signal(e1)

on update price('Blue', B) if price('Soft', S) and $B_{old} > S$ and $B_{new} \leq S$ then signal(e2)

on update price('Soft', S) if price('Blue', B) and $B > S_{old}$ and $B \leq S_{new}$ then signal(e2)

on not e2 in [e1, e1 + 5hours] then send trade(buy, 'Blue', 40000)

Suppose the broker cannot carry out the instruction to buy 40,000 shares, perhaps because there are none for sale. This can be be handled by a rule that is triggered on an *orderFailed* event. For example, we could test to see if the price of 'Blue' shares is still greater than 'Soft' shares and stays that way for another five hours as follows:

on orderFailed('Blue') if price('Blue', B) and price('Soft', S) and $B > S$ then signal(e1)

4.2.2 An Agent System Solution

For simplicity, we will use here a notation for plans that is as similar as possible to that used for the triggers and rules above, introducing the `while` keyword for maintenance conditions. The event predicate `addbel` is true when a tuple is added to a relation. The event predicate `modbel` is true when the value of a tuple in a relation changes. Its parameters are the previous and current values of the tuple. A suitable set of plans would be:

on modbel(price('Blue', $\mathtt{B}_{old}$), price('Blue', $\mathtt{B}$)) if price('Soft', $\mathtt{S}$) and $\mathtt{B}_{old} \leq \mathtt{S}$ and $\mathtt{B} > \mathtt{S}$
  while price('Blue', $\mathtt{B}_{new}$) and price('Soft', $\mathtt{S}_{new}$) and $\mathtt{B}_{new} > \mathtt{S}_{new}$
  then waitfor elapsed(5hours) andthen send trade(buy, 'Blue', 40000)

on modbel(price('Soft', $\mathtt{S}_{old}$), price('Soft', $\mathtt{S}$)) if price('Blue', $\mathtt{B}$) and $\mathtt{B} \leq \mathtt{S}_{old}$ and $\mathtt{B} > \mathtt{S}$
  while price('Blue', $\mathtt{B}_{new}$) and price('Soft', $\mathtt{S}_{new}$) and $\mathtt{B}_{new} > \mathtt{S}_{new}$
  then waitfor elapsed(5hours) andthen send trade(buy, 'Blue', 40000)

on addbel orderFailed('Blue')
  while price('Blue', $\mathtt{B}_{new}$) and price('Soft', $\mathtt{S}_{new}$) and $\mathtt{B}_{new} > \mathtt{S}_{new}$
  then waitfor elapsed(5hours) andthen send trade(buy, 'Blue', 40000)

4.3 Trade Wars

Suppose 'Blue' has two ways to deal with 'Soft' undercutting the price of 'Blue' widgets; either buy a controlling share of 'Soft', or lower the price of its widgets. Either of these ways may be suitable in the circumstances. In an AS, one could specify that both plans are triggered by the event of 'Soft' undercutting the price of 'Blue' widgets, and write a metalevel plan to actually choose between the two. The default interpreter behaviour is to try the first plan and, only if it fails — for example 'Blue' does not control 'Soft' after 2 days — to try the second plan.

4.3.1 An Active Database Solution

Support for metalevel programming is not often provided in ADBs. Hence, if it is required that the first method be tried before the second one, then that decision must be encoded in the triggers themselves:

on update widgetPrice('Soft', $\mathtt{S}$)
  if widgetPrice('Blue', $\mathtt{B}$) and $\mathtt{S}_{old} > \mathtt{B}$ and $\mathtt{S}_{new} < \mathtt{B}$ then signal(underCut)
on underCut
  if ownsPercent('Soft', X) and $X \leq 50$ and issuedShares('Soft', Y) and
    $Z = Y \times (51 - X) \div 100$ then send trade(buy, 'Soft', Z)
on underCut before update ownsPercent('Soft', X)
  if $X > 50$ then send terminate('Soft') andthen signal(controlsSoft)
on not controlsSoft in [underCut, underCut + 2days]
  if widgetPrice('Soft', $\mathtt{S}$) and $\mathtt{B} = 0.9 \times \mathtt{S}$
  then send terminate('Soft') andthen update widgetPrice('Blue', $\mathtt{B}$)

By extending this example, we can highlight another feature of AS plan languages that can make them more convenient than the action languages typical of ADB systems. Observe that 'Blue' can gain control of 'Soft' by either buying 'Soft' shares or by buying a controlling fraction of companies that own shares in 'Soft'. 'Blue' may even have to sell other shares to finance this takeover. Computing what shares 'Blue' should sell and buy to gain control of

'Soft' could actually turn out to be quite a lengthy computation. Indeed, the price of 'Soft' widgets may rise above those of 'Blue' widgets before the system has actually worked out what shares to buy! In this case, the system should stop the computation.

Once again, this could be quite difficult to implement in a natural way in an ADB. One would need to make sure that the computation of what shares to buy was not implemented as one atomic operation, but instead as a series of operations with an explicit check on widget prices in between each operation. We will not even attempt to give an ADB solution to this extended problem.

4.3.2 An Agent System Solution

We specify a maintenance condition to ensure that the plan is terminated if 'Soft' stops undercutting the 'Blue' widget price. The lengthy computation appears as a subgoal `computeOrder` which returns a list of buy/sell orders as output. Note that the body of this plan contains a conditional action with the structure `when` *condition* `do` *action* `else` *action*.

on modbel(widgetPrice('Soft', $\mathtt{S}_{old}$), widgetPrice('Soft', $\mathtt{S}$))
  if widgetPrice('Blue', $\mathtt{B}$) and $\mathtt{B} \leq \mathtt{S}_{old}$ and $\mathtt{B} > \mathtt{S}$ and
    ownsPercent('Soft', X) and $X \leq 50$ and issuedShares('Soft', Y)
  while widgetPrice('Blue', $\mathtt{B}_{new}$) and widgetPrice('Soft', $\mathtt{S}_{new}$) and $\mathtt{B}_{new} > \mathtt{S}_{new}$
  then achieve computeOrder(X, Y, List) andthen send trade(List) andthen
    waitfor ((ownsPercent('Soft', U) and $U > 50$) or elapsed(2days)) andthen
    send terminate(List) andthen
    when test (ownsPercent('Soft', V) and $V > 50$)
      do signal(controlsSoft)
      else test widgetPrice('Soft', W) andthen update widgetPrice('Blue', $W \times 0.9$)

4.4 Atomicity and Concurrency

The absence from ASs of certain features usually found in ADBs can make them less convenient for some applications. Suppose that our stock-broker sends us messages whenever share transactions are completed. This will require a trigger to ensure that the database keeps derived relations, such as share ownership percentage, appropriately updated:

on brokerMessage(Name, Qty)
  if ownsPercent(Name, OldPC) and issuedShares(Name, Total) and
    $\text{NewPC} = ((\text{OldPC} \times \text{Total} \div 100) + \text{Qty}) \div 100$
  then update ownsPercent(Name, NewPC)

The look-up of the old value of the percentage of shares owned, the computation of the new percentage, and the update to the new value all need to be done in one atomic operation to guarantee that multiple updates to the same value do not result in any updates being lost.

In an ADB, the underlying transaction support would ensure that multiple updates to the same value happen sequentially, while updates to different values can happen concurrently. Currently, most ASs would offer no support for this, requiring that it be handled explicitly by the update plans.

5 Further Issues

5.1 Responsiveness

Agent systems are typically designed to support time critical applications such as manufacturing/process control and air traffic control. In these systems it is necessary to impose constraints on the time taken to select and execute plans, and to guarantee that events are noticed sufficiently promptly [22]. One may need to choose a plan based on the criterion of how long it will take to execute as well as the quality of the result it produces — a sub-optimal solution within the available time is better than an optimal solution delivered too late to be useful. Complex metalevel reasoning may be needed to make this choice, but this too must occur in bounded time. Furthermore, once a plan has been chosen, it may have to be abandoned due to a change in the agent's beliefs.

For example, in an air traffic control application, a particular agent's task may be to find an optimized schedule for plane landings. If more planes enter the monitored airspace, or weather conditions change, then a current intention to produce a schedule may need to be abandoned or restarted to accommodate this change in beliefs. The absence of facilities for controlling responsiveness means that these kind of dynamically changing scenarios cannot be easily supported by an ADB.

5.2 Success and Failure

ASs allow one to evaluate the success/failure of actions, but it is not as easy to find out the status of an ADB action after execution. The interpreter in an AS behaves in a way that takes into account the possibility of such failures; alternative paths within a plan will be tried if actions or subgoals fail, and alternative plans will be tried on plan failure. These characteristics are essential for robust embedded applications. While such behaviour can be implemented within an ADB, it is not directly supported by the execution engine.

5.3 Persistence of Intentions

Another salient feature of ASs is the long running (possibly perpetual) nature of intentions. An ADB can achieve similar behaviour through the use of an embedded application — one which interacts with the database system at regular intervals and issues the appropriate (trans)actions depending on the feedback it receives. This approach, however, can lead to inefficiencies and is less desirable from a software engineering point of view.

Another approach might be to use an active database with deductive capabilities (and hence the possibility of persistence through infinite recursion) to provide similar functionality. A different solution would be to have a complex event algebra which permitted expressions that referred to past events stored by a history mechanism.

5.4 Rollback and Recovery

For performance reasons, the belief database of a BDI agent is often implemented in main memory, and only intermittently checkpointed to backing store, if at all. Usually there is no support for examining the update history. ADBs, by contrast, tend to regard the backing store as primary, and expend considerable effort to guarantee its integrity, to support rollback of transactions, and recovery to previous database states. The development of "industrial strength" applications will require the integration of such techniques into the next generation of ASs.

5.5 Metalevel Constructs

Metalevel constructs are not usually provided by ADBs, whereas they are a valuable feature of ASs. Suppose an event occurs in an ADB that triggers two rules whose simultaneous execution would interfere. For example, the following rules will clearly interfere if executed simultaneously:

on update price('Blue', $\mathtt{B}$)
  if price('Soft', $\mathtt{S}$) and $\mathtt{B}_{old} \geq \mathtt{S}$ and $\mathtt{B}_{new} < \mathtt{S}$
  then send trade(sell, 'Blue', 10000)

on update price('Blue', $\mathtt{B}$)
  if price('Moon', $\mathtt{M}$) and $\mathtt{B}_{old} \leq \mathtt{M}$ and $\mathtt{B}_{new} > \mathtt{M}$
  then send trade(buy, 'Blue', 10000)

One approach to avoiding this problem is to hard-code logic into the rule manager, which may have a policy of always selecting the highest priority rule (e.g. by a textual ordering). Another method is to encode the necessary checking into the rules themselves so that they are mutually exclusive.

A more elegant way of handling the problem is to allow the system designer to resolve the conflict by defining a metalevel rule to make the selection. Whenever both rules are activated simultaneously, an event is generated that triggers another (higher priority) rule whose execution determines which rule is to be executed. This is the approach adopted in ASs. It is attractive because it allows localization of the rule selection strategy; one only has to alter the metalevel plan as opposed to changing the original rules or recompiling the rule manager.

ASs may also generate metalevel events to allow selection between different instances of the same plan, between different execution paths within a plan, to determine the order in which simultaneous events are processed, and to control the order of execution of concurrent intentions.

6 Research Issues and Future Directions

An obstacle to the success of both technologies is the lack of coherent system and rule/plan design methodologies. For example, in an ADB it is unclear how much functionality should be performed by the ECA triggers and how much by external applications. An inappropriate choice may incur performance penalties. Likewise, in ASs one may write complex standalone plans or else construct simple plans with a sophisticated metalevel reasoner to control them. Another issue is determining what agents are needed to solve a particular problem — this is similar to schema design in an ADB. One possibility is the adaptation of object-oriented design methodologies [30, 4] to the design of ASs; this is currently an active research area [24].

A question related to rule design is the possibility of reasoning about rule sets and testing them for certain properties. Active database research has examined properties such as confluence and termination [2], albeit in a restricted way. For agent systems these properties may be less relevant, since intentions can be persistent and hence non-terminating, but there is still much to be gained from such efforts. Another, more general approach is to look at the externally observable behaviour of the system as a whole. Does the addition of extra rules destroy the correctness of existing rule sets, or perhaps degrade system performance unacceptably? Tractable theories that address these issues will be necessary if we are to construct robust, maintainable, large-scale systems.

Another issue is the integration of the two paradigms (see [9] for an example of using both technologies co-operatively). For example, one could incorporate transactions into an AS to provide recovery and robustness. Alternatively, one could incorporate long lived processes and their associated maintenance conditions into an ADB. The purpose of these modifications is not to replace one technology with the other, but to enhance the functionality of the respective systems. Indeed these technologies have distinct application domains, one being data intensive and the other being process intensive.

Other interesting notions are to base an agent system upon an active database, or to include agents within an active database. These ideas raise important issues about the granularity of agents, but are likely to be impractical for serious use, however, until the state of the art in debugging techniques advances. The difficulties involved in verifying the correctness of concurrent systems are substantial and the coupling of both of these technologies would exacerbate the problem.

It is in the area of language syntax and semantics that we feel ASs and ADBs have the most to offer each other. As yet, there is no consensus as to what language features should be standard in either, thus there is plenty of scope for crossover of ideas. In particular, ASs could profitably adopt more powerful event languages and ADBs could utilize more plan-like action specifications. In the same way, optimization techniques such as incremental condition evaluation for ADBs and the various compilation-time optimizations employed in ASs could be utilized. Finally, real world ASs will need to adopt or adapt some of the history, rollback and recovery techniques employed in typical ADBs.

Despite the challenges presented, the rewards offered by (re)active and goal-directed functionality are great. We hope that as each field matures, researchers remain conscious of their essentially complementary nature.

References

[1] Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI-87*, pages 268–272, Seattle, WA, 1987.

[2] A. Aiken, J. Widom, and J. M. Hellerstein. Behavior of database production rules: Termination, confluence and observable determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1992.

[3] J. A. Bailey and K. Ramamohanarao. Issues in active databases. In *Proceedings of the Sixth Australasian Database Conference*, pages 27–35, Glenelg, South Australia, 1995.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 2nd edition, 1994.

[5] Michael E. Bratman. *Intentions, Plans, and Practical Reason.* Harvard University Press, Cambridge, MA, 1987.

[6] B. Burmeister and K. Sundermeyer. Cooperative problem-solving guided by intentions and perception. In *Proceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds, MAAMAW '91*, Kaiserslautern, Germany, 1991.

[7] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Putting active databases to work. In *IEEE Data Engineering*, 1992.

[8] S. Ceri and J. Widom. Deriving incremental production rules for deductive data. *Information Systems*, 19(6), 1994.

[9] S. Chakravarthy, K. Karlapalem, S. B. Navathe, and A. Tanaka. Database supported cooperative problem solving. Technical Report UF-CIS-TR-92-046, University of Florida, Gainesville, Florida, 1992.

[10] Paul R. Cohen, Michael L. Greenberg, David M. Hart, and Adele E. Howe. Real-time problem solving in the phoenix environment. In *Proceedings of the Workshop on Real-Time Artificial Intelligence Problems at IJCAI-89*, 1989.

[11] Paul R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

[12] U. Dayal, E. Hanson, and J. Widom. Active database systems. In *Modern Database Systems: The Object Model, Interoperabilty and Beyond.* Addison Wesley, Reading Massachusetts, 1994.

[13] Keith Decker and Victor Lesser. Analyzing the need for meta-level communication. In *Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence, DAI '93*, Hidden Valley, Pennsylvania, 1993.

[14] R. J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Department of Computer Science, Yale University, New Haven, Connecticut, 1989.

[15] R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence, AAAI-87*, pages 202–206, Seattle, WA, 1987.

[16] Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. In *Proceedings of the IEEE Special Issue on Knowledge Representation*, volume 74, pages 1383–1398, 1986.

[17] M.P. Georgeff and F.F. Ingrand. Real-time reasoning: The monitoring and control of spacecraft systems. In *Proceedings of the Sixth IEEE Conference on Artificial Intelligence Applications*, Santa Barbara, California, 1990.

[18] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *Transaction on Database Systems*, 1983.

[19] E. Hanson et al. Flexible and recoverable interaction between applications and active databases. Technical Report UF-CIS-TR-94-033, University of Florida, Gainesville, Florida, 1994.

[20] J. V. Harrison and S. W. Dietrich. Incremental view maintenance. In *Proceedings of the Fifth Australasian Database Conference*, Christchurch, New Zealand, 1994.

[21] F. Hayes-Roth, B., R. Washington, R. Hewett, M. Hewett, and A. Seiver. Intelligent monitoring and control. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89*, pages 243–249, Detroit, MI, 1989.

[22] Felix Ingrand and Michael P. Georgeff. Managing deliberation and reasoning in real-time AI systems. Technical Report 10, Australian Artificial Intelligence Institute, Melbourne, Australia, 1990.

[23] David Kinny and Michael Georgeff. Commitment and effectiveness of situated agents. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 82–88, Sydney, 1991.

[24] David Kinny and Michael Georgeff. A design methodology for BDI agent systems. Technical Report 55, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.

[25] David Kinny, Magnus Ljungberg, Anand Rao, Elizabeth Sonenberg, Gil Tidhar, and Eric Werner. Planned team activity. In *Proceedings of the Fourth European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '92*, page 20, Viterbo, Italy, 1992. Also appears as Australian Artificial Intelligence Institute Technical Note 31, Melbourne, Australia, 1992.

[26] Nils J. Nilsson. Towards agent programs with circuit semantics. Technical Report STAN-CS-92-1412, Stanford University, Stanford, CA, 1992.

[27] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, KR '91*, pages 473–484, Cambridge, MA, 1991.

[28] Anand S. Rao and Michael P. Georgeff. An Abstract Architecture for Rational Agents. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning, KR '92*, pages 439–449, Boston, MA, 1992.

[29] S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In J. Y. Halpern, editor, *Proceedings of the First Conference on Theoretical Aspects of Reasoning about Knowledge*, San Mateo, CA, 1986. Morgan Kaufmann Publishers.

[30] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliifs, NJ, 1991.

[31] Yoav Shoham. Agent0: A simple agent language and its interpreter. In *Proceedings of the Ninth National Conference on Artificial Intelligence, AAAI-91*, pages 704–709, San Jose, CA, 1991.