Agent-Based Process Management

John Debenham
Faculty of Information Technology, University of Technology, Sydney
debenham@it.uts.edu.au

Abstract. A categorisation of business process covers the full business process spectrum from routine production workflows to high-level emergent processes. The first of these categories is activity-driven processes; they are managed by a single reactive agent architecture. The second of these categories is goal-driven processes; they are managed by a multiagent system. The third of these categories is knowledge-driven processes; they may not be managed as such. Two agent-based architectures manage activity-driven and goal-driven processes respectively.

INTRODUCTION

Business process management is an established application area for multiagent systems (Jennings, et al, 2000). The term *business process* here covers the spectrum from production workflow to emergent process (Dourish, 1998). *Production workflows* are well-defined and highly repetitive processes. *Emergent processes* are processes that are not necessarily pre-defined, that may not be of a routine nature and that may rely on some level of initiative from the system to bring them to a conclusion. The automated aspects of a business process are managed by a *process management system* that applies a sequence of *activity instances* to each *process instance*.

Processes across the workflow / emergent process spectrum have differing management requirements. A categorisation of processes is given into three categories. Together they cover the full process spectrum. The direction of a *knowledge-driven process* is determined in part common sense and environmental knowledge; so they can not be managed, at best they may be supported. The management of *activity-driven processes* is achieved with a single reactive agent architecture. The management of *goal-driven processes* is achieved with a multiagent system based on a three-layer BDI (Belief-Desire-Intention) hybrid agent architecture.

Systems for managing activity-driven and goal-driven processes use virtual web documents that may be interactive. In both systems it is possible to work remotely—all that is required is an Internet connection and a PC. One of the design constraints on the systems was that it should be possible to use them using a laptop and the telecommunication lines provided in many commercial aircraft.

THREE PROCESS CATEGORIES

The terms "production workflow" and "emergent process" are types of business process and are usually defined in loose terms. An alternative categorisation of business process is given here by defining three categories of process in terms of their process management requirements.

Following (Fischer, 2000) a *business process* is "a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships". Implicit in this definition is the idea that a process may be repeatedly decomposed into linked sub-processes until those sub-processes are "activities" which are atomic pieces of work. [viz. (op.cit) "An *activity* is a description of a piece of work that forms one logical step within a process." and "An activity typically generates one or more work items which together constitute the *task* to be undertaken by the user."]. In general this decomposition will contain conditional branches. So the decomposition of a process may be represented as a tree where the nodes are labelled with the names of activities and the arcs are labelled with conditional expressions. Repetition is permitted and so these trees may be unbounded. Each process and sub-process has a process *patron* who is responsible for that process; this responsibility may be delegated. Each process has a *goal* that is a state that the process intends to achieve. Each process has a *termination condition* that determines when that process should cease; the termination condition *may* be related to the process' goal. This definition of a business process is based on the assumption that such a decomposition will always work. This assumption is valid for production workflows. But for more sophisticated business processes, one decomposition may work well for a while and then, for reasons that are not understood *within* the system, may fail. So *some* processes are associated with a unique, valid decomposition.

Three categories of business process cover the business process spectrum. Activity-driven processes are the simplest category to manage. They are typically handled by clerical staff. This category includes production workflows. Knowledge-driven processes are the most complex. They are typically handled by middle and senior staff and so represent

Table 1. Properties of the three categories of process

	Task-driven	Goal-driven	Knowledge-driven
Process goal	Determined by process patron, remains fixed	Determined by process patron, remains fixed	Determined by process patron, may mutate
Process termination condition	Process goal achieved	Process goal achieved	Determined by process patron
Next goal	Determined by instance history	Determined by instance history	Determined by process patron
Next task	Determined by instance history and next goal—should achieve next goal	Chosen (somehow) on the basis of instance history and next goal—may not achieve next goal	Chosen by process patron to generate process knowledge.
Next activity termination condition	Next goal achieved	Next goal achieved, if it fails then try another way	Determined by process patron

the upper end of business process management. The three categories of business process are:

- An *activity-driven process* can be associated with a—possibly conditional—sequence of activities such that execution of the corresponding sequence of tasks "always" achieves the process goal. Each of these activities has a goal, and is associated with a task that on its termination "always" achieves this goal. Production workflows are often activity-driven processes.
- A *goal-driven process* has a process goal, and can be associated with a—possibly conditional—sequence of sub-goals such that achievement of this sequence "always" achieves the process goal. Achievement of a sub-process goal is the termination condition for that sub-process. Each of these sub-goals is associated with at least one activity and so with at least one task. Some of these tasks may work better than others, and there may be no way of knowing which is which. A task for an activity may fail outright, or may be otherwise ineffective at achieving its goal. In other words, unpredictable task failure is a feature of goal-driven processes. If a task fails then another way to achieve its sub-goal may be sought.
- A *knowledge-driven process* may have a process goal, but the goal may be vague and may mutate (Dourish, 1998). Mutations are determined by the process patron, often in the light of knowledge generated during the process. The termination condition for a knowledge driven sub-process is not necessarily related to the achievement of the sub-process goal. At each stage in a knowledge-driven process instance the "next goal" is chosen by the process patron; this choice is made using general knowledge about the context of the process—called the *process knowledge* which may be used to direct knowledge-driven processes. Unfortunately the process knowledge is far too complex to represent in general. For this reason knowledge-driven processes may not be managed.

Properties of the three categories of process are shown in Table 1.

ACTIVITY-DRIVEN PROCESS

An activity driven process can be associated with a—possibly conditional—sequence of activities such that execution of the corresponding sequence of tasks "always" achieves the process goal. The idea behind activity-driven processes is that process failure will not happen. A workflow application in a draconian organisation may be seen to be failure-proof, and so could be treated as activity-driven. In practice, even production workflow applications can fail; a clerk can "click the wrong button" for example. So here the reactive part of the agents' architecture is used for low level operations only.

Given an activity-driven process, construct a node labelled with the activity that creates that process. From that node directed arcs lead to other nodes labelled with activities so that every possible sequence of activities that leads to a node that destroys the process is represented. If more than one arc follows a node then those arcs are labelled with the condition under which they should be followed. No arcs lead from a node that destroys a process. Then re-label the arcs as $\alpha(C) / D$ where α is the event that the activity that precedes the arc has terminated, C is the arc condition if any, and D is the actions that the management system should perform prior to the activity that follows the arc. In this way activity-driven processes are represented as statecharts see Fig. 1.

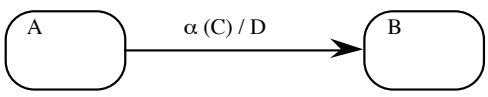

Fig. 1. Statechart for activity-driven process

Some of what a web-based process management system has to do is to add or delete pointers to virtual documents to or from the users work area.

Operations of this sort are represented as actions D on the state chart.

Some of what a web-based process management system has to do is to add or delete pointers to virtual documents to or from the users' "In Trays". Operations of this sort are represented as actions on a state chart.

The resulting statechart has a natural implementation as event-condition-action state-transition rules of the form:

if in state A and event α occurs
and condition C is true
then perform action D and enter state B

So activity-driven process management can be effected using a single reactive agent, or expert system, containing rules of this form. If the rules in such a knowledge base are indexed by their "from state" then the maintenance of the knowledge base is quite manageable.

The state label can be quite complex. For example, a state label for a process that is to be circulated amongst n people, two at a time, until some event occurs can be represented as an $n \times 2$ matrix.

Assessment. The principal strength of the activity-driven system is its simplicity. The system itself is just an interpreter for rules of the form described above. The interpreter was written from scratch in Java. It operates by maintaining a virtual "In Tray" for each user represented as an HTML page. Items on this page contain messages, which are typically quite simple, and references to the virtual web documents that are part of the process instance. If the application is a production workflow problem, and if there is not a significant number of process constraints, then modelling the application and statecharts is usually a simple and natural business. The subsequent representation of this statechart in the system is then a trivial matter.

One shortcoming of the activity-driven system is that the central system takes responsibility for managing each process instance. That is, the central system is the process patron. This poses no theoretical problem but it does mean that if processes are time (or cost) constrained then the state chart representation has to cope with these constraints. State chart representations that do not include time or cost constraints can be very neat, but when these constraints are included to allow for every possible constraint violation the charts can become an appalling mess. In goal-driven process the responsibility for process instances is explicitly delegated, and so too the responsibility for complying with instance constraints. This has the effect of simplifying the constraint management problem for two reasons. First the goal-driven formalism is plan-based and—unlike statecharts—includes the representation of failure in the formalism. Second the distribution of the constraint management problem to the component agents in the system scales this problem down.

Another shortcoming of the activity-driven system stems from the fact that it is a centralised system managing an essentially distributed task. For example, if the personal preferences of each user are to be represented in a centralised system then this leads to a complex maintenance problem—particularly if individual personal preferences are in conflict. In the goal-driven system described below, conflicts of this form are dealt with naturally through the inter-agent negotiations.

GOAL-DRIVEN PROCESS

A *goal-driven process* has a process goal, and can be associated with a—possibly conditional—sequence of sub-goals such that achievement of this sequence "always" achieves the process goal. Goal-driven process is *like* activity-driven process in that for each activity each task is *intended* to realise the sub-goal of that activity. Goal-driven processes are *unlike* activity-driven processes in that execution of these tasks is unpredictable. What has worked well for some time may now fail to work at all. Further the reason for failure may lie outside the understanding of the system.

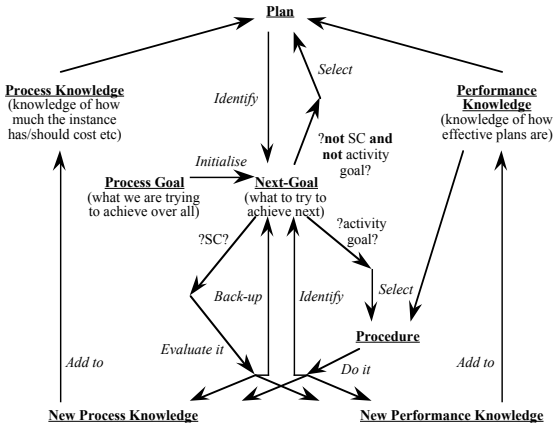

Fig. 2. Goal-driven process management (simplified view)

Goal-driven processes may be modelled as state and activity charts (Muth, et al, 1998). The primitives of that model are activities and states. An *activity chart* specifies the data flow between activities. An activity chart is a directed graph in which the arcs are annotated with data items. A *state chart* is a representation of a finite state machine in which the transitions annotated with event-condition-action rules; see Fig. 1. (Muth, et al, 1998) show that the state and activity chart representation may be decomposed to pre-empt a distributed implementation. Each event on a state chart may be associated with a goal to achieve that event, and so a state chart may be converted to a plan whose nodes are labelled with such goals. Unlike activity-driven processes, the success of execution of a plan for a goal-driven process is not necessarily related to the

achievement of its goal. One reason for this is that an instance may make progress outside the process management system—two players could go for lunch for example. That is, when managing goal-driven processes there may be no way of knowing the "best" task to do next. Each high-level plan for a goal-driven process should terminate with a check of whether its goal has been achieved. To represent goal-driven processes, a form of plan is required that can accommodate failure. This is discussed below.

Fig. 2 shows a simplified view of the management of goal-driven processes. The primitives shown in Fig. 2 are goals and plans. Some goals are associated with *executable* activities and so with tasks. If a goal is not associated with an activity then it should be the subject of at least one plan. Fig. 2 presents a simplified view because a sub-goal of a goal-driven process goal will not necessarily be goal-driven, and because that Figure does not show that plans may be aborted—as discussed below.

So goal-driven process management has a requirement *both* for a software architecture that can cope naturally with failure, *and* for some technique for intelligently selecting which is the "best" task to do next (Debenham, 2000). Any general-purpose architecture can achieve this first requirement but the process architecture described below is particularly suitable.

System Architecture. In the goal-driven process management system an agent supports each (human) user. These agents manage their users' work and manage the work that a user has delegated to another user/agent pair. Sub-process *delegation* is the transfer of responsibility for a sub-process from one agent to another. A *delegation strategy* decides who to give responsibility to for doing what. Delegation strategies in manual systems can be quite elementary; delegation is a job that some humans are not very good at. A user of the system may specify the delegation strategy and may permit her agent to delegate for her, or may delegate manually. In doing this, the user has considerable flexibility first in defining payoff and second in specifying the strategy itself. A delegation strategy may attempt to balance some of the three conflicting principles: maximising payoff, maximising opportunities for poor performers to improve and balancing workload.

The goal of this system is to manage goal-driven processes. The systems architecture consists of one agent for each (human) user; the role of each agent that of an assistant to its user. The components of each *node* in this system are illustrated in Fig. 3. The user interacts with a virtual work area and a virtual diary. The *work area* contains three components which are: the process instances awaiting the attention of the user, the process instances for which the user has delegated responsibility to another agent, and the process instances that the agent does not understand. The *diary* contains the scheduled commitments of the user. The agent manages the work area and may also interact with the diary.

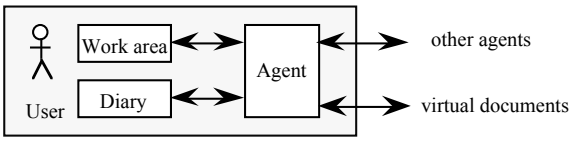

Fig. 3. A system node in the multiagent system

The basis for the interaction protocol is open cooperation; the agents are truthful with one another about the contents of their users' diaries—for example. Each agent manages the work of its user in the virtual work area and deals with the delegation of responsibility for sub-processes to other nodes. This delegation is achieved by inviting a selected set of nodes to bid for work. A bid from a node contains information on the firm and preferred constraints that that user presently has, and information about that user's work—including an estimate of the cost, in time, that that user may incur.

The conceptual architecture of the agents belongs to a well-documented class. Wooldridge describes a variety of architectures (Ch. 1 in (Weiss, 1999)). One well-documented class of hybrid architectures is the three-layer, BDI agent architectures. One member of this class is the INTERRAP architecture (Müller, 1996), which has its origins in the work of (Rao and Georgeff, 1995). In the goal-directed process management system, the agent's conceptual architecture differs slightly from the INTERRAP conceptual architecture; it is intended specifically for business process applications. This conceptual architecture is shown in Fig. 4. It consists of a three-layer BDI architecture together with a *message area*. A *message manager* manages the message area. Access to the message area is available to other agents in the system who may post messages there and, if they wish, may remove messages that they have posted. The idea behind the message area is to establish a persistent part of the agent to which the other agents have access. This avoids other agents tampering directly with an agent's beliefs, and enables agents to freely remove their messages from a receiving agent's message board if they wish. The message area is rather like a person's office "in-tray" into which agents may place documents, and from which they may remove those documents if they wish. The agents' world beliefs are derived *either* from reading messages received from a user, *or* from reading the documents involved in the process instance, *or* from reading messages in the message area. These activities are fundamentally different in that documents are "passive"; they are read only when information is required. Users and other agents send messages when they feel like it. Beliefs play two roles. First, they may be partly or wholly responsible activating a local or cooperative trigger that leads to the agent committing to a goal, and may thus initiate an intention (eg. a plan to achieve what a

message asks, such as "please do xyz"). This is part of the *deliberative reasoning* mechanism. Second, they can be partly or wholly responsible for activating a reactive procedure trigger that, for example, enables the execution of an active plan to progress. This is part of the *reactive reasoning* mechanism.

The *control architecture* is essentially to the INTERRAP control architecture. In outline, the deliberative reasoning mechanism employs the non-deterministic procedure: "on the basis of current *beliefs*—identify the current *options*, on the basis of current options and existing commitments—select the current commitments (or *goals*), for each newly-committed goal choose a *plan* for that goal, from the selected plans choose a consistent set of things to do next (called the agent's *intentions*)". If the current options do not include a current commitment then that commitment is dropped. So if agent A sends agent B a message M asking agent B to do something, agent B may commit to do this. If agent A then removes M from B's message area then at B's next deliberative cycle B should decommit to that task.

The reactive reasoning mechanism takes precedence over the deliberative reasoning mechanism. The *reactive frequency* is the frequency at which an attempt is made to fire all active reactive triggers. The reactive frequency is thirty seconds. The *deliberative frequency* is the frequency at which the deliberative reasoning mechanism is activated. To maintain some stability in each user's work area, the deliberative frequency is five minutes.

KQML (Knowledge Query and Manipulation Language) is used for inter-agent communication (Finin et al, 1997). If agent A wishes to tell something to agent B then it does so by posting a message to agent B's message area. Each message contains an instruction for the message manager. Two such instructions are:

- post message and remove on condition—the sender agent is asking the receiving agent's message manager to display a message in the receiving agent's message area *until* the stated condition is satisfied, and
- remove message—the sender agent is asking the receiving agent's message manager to remove one of the sender's previous messages from the receiving agent's message area.

Inter-agent communication has been implemented as socket-to-socket connections on the local network and email (using addresses known only to the agents) so that a user may continue to use the system remotely—even from an aeroplane.

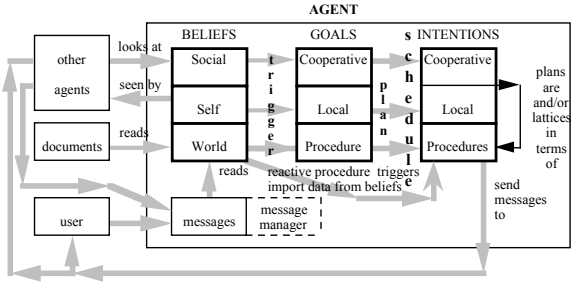

Fig. 4. Conceptual architecture

Deliberative Reasoning. The form of plan is slightly more elaborate than the form of agent plan described in (Rao and Georgeff, 1995) where plans are built from single-entry, triple-exit blocks. Those three exits represent success, failure and abort. Powerful though that approach is, it is inappropriate for process management where whether a plan has executed successfully is not necessarily related to whether that plan's goal has been achieved.

In goal-driven process management applications a plan can not necessarily be relied upon to achieve its goal even if all of the sub-goals on a chosen path through the plan have been achieved. On the other hand, if a plan has failed to execute then it is possible that the plan's goal may still have been achieved. So, a necessary sub-goal in every high-level plan body is a sub-goal called the "success condition". The *success condition* (SC) is a procedure whose goal is to determine whether the plan's goal has been achieved. The success condition is the final sub-goal on *every* path through a plan. The success condition is a procedure; the execution of that procedure may succeed (✓), fail (✗) or abort (**A**). If the execution of the success condition fails then the overall success of the plan is unknown (**?**). So the four possible plan exits resulting from an attempt to execute a plan are as shown in Fig. 5.

A plan body is represented as a directed AND/OR graph, or state-transition diagram, in which some of the nodes are labelled with sub-goals. The *plan body* may contain the usual conditional constructs such as **if...then**, and iteration constructs such as **while..do..**. The diagram of a plan body has one start state (activation condition "ac", and activation action "α"), and stop states either labelled as success states "✓" (success action "σ"), fail states "✗" (fail action "ϕ"), unknown states "**?**" (unknown action "υ") or abort states "**A**" (abort condition "ab", and abort action "ω").

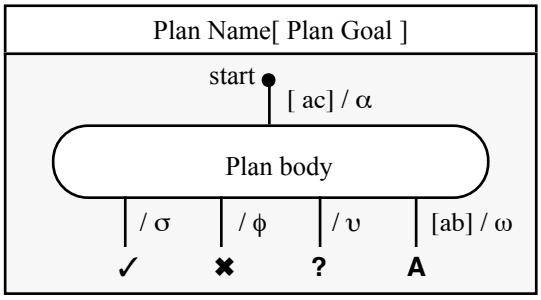

Fig. 5. The process agent plan

Reactive Reasoning. Reactive reasoning play two roles: first, if a plan is aborted then its abort action is activated; second, if a procedure trigger fires then its procedure is activated—this includes hard wired procedure triggers that deal with urgent messages such as "the building is on fire!". Of these two roles the first takes precedence over the second.

Reactive reasoning is achieved by rules of the form:

if <trigger state> **and** <belief state> **then** <action> **and** <trigger state>

where the <trigger state> is a device to determine whether the trigger is active or not, and <belief state> is something that the agent may believe; <action> may be simply to transfer some value to a partly executed plan, or may be more profound such as to abort a plan and decommit a goal.

Each plan contains an optional abort condition [ab] as shown in Fig. 5. These abort conditions are realised as procedural *abort triggers* that are activated when their plan is active. Active abort triggers scan the agent's beliefs for the presence of their abort condition. Abort triggers are only active if the goal of the plan to which they are attached is a goal that the agent is presently committed to achieving. If a plan is aborted then any active sub-goals of that plan are also aborted.

If an agent A has an active plan P that requires input from its user or another agent B then a procedure sends a request message directly to B with a unique identifier #I, and a reactive procedure trigger is activated (ie. made "active"):

if active **and** believes B's response to #I is Z **then** pass Z to P **and** not active

In this way 'data' is passed to partly executed plans using reactive triggers. Reactive triggers of this form are associated with belief states of the form "B's response to #I is known". Such a procedure trigger is active when its associated sub-goal is committed to but has not been realised.

The abort triggers have a higher priority than reactive triggers. So if a plan's abort trigger fires and if an active sub-goal in that plan is the subject of a reactive trigger then that sub-goal will be deactivated so preventing that reactive trigger from firing even if the required belief is present in the world beliefs.

Selection and Delegation. For goal-directed processes, there may be no way of knowing what the "best" thing to do next is, and that next thing may involve delegating the responsibility for a sub-process to another agent. This raises two related issues. The first issue is *selection*; that is, given a goal select the "best" plan or activity for that goal. The second issue is *delegation*; that is, the problem of deciding whom to ask to take responsibility for what and then following up on the delegated responsibility to make sure that the work is done. The sense in which "best" is used here does *not* mean that selection and delegation are optimisation problems. A process management system is one *part of* an organisation. Ideally the goal of a process management system should be that of its organisation, such as "to maximise corporate profits". But, unless measurements on the full range of corporate activity are available to it, the management system is unable to address such a global goal. On the other hand, attempts to optimise the performance of the process management system only can lead, for example, to over use of the best performing staff. So if the only measurements available are derived from within the process management function then the meaning of "best" should take note of global implications, such as the equity in the working environment, as well as the quality of the process output. An attempt to define what is meant by "best" in functional process management terms that attempts to address corporate priorities may lead to conflicting principles such as: maximising payoff, providing opportunities for poor performers to improve and balancing workload.

To deal with selection and delegation, performance knowledge is gathered, as is illustrated on Fig. 2. The *performance knowledge* comprises performance statistics on the operation of every plan and activity. In the case of a parameter, p, that can reasonably be assumed to be normally distributed, an estimate for the mean of p, μ_p, is revised on the basis of the i'th observation ob_i to:

$$\mu_{p_{new}} = (1 - \alpha) \cdot ob_i + \alpha \cdot \mu_{p_{old}}$$

which, given a starting value $\mu_{p_{initial}}$, and some constant α, $0 < \alpha < 1$, approximates the geometric mean of all observations to date. In the same way, an estimate for $\sqrt{2/\pi}$ times the standard deviation of p, σ_p, is revised on the basis of the i'th observation ob_i to:

$$\sigma_{p_{new}} = (1 - \alpha) \cdot | ob_i - \mu_{p_{old}} | + \alpha \cdot \sigma_{p_{old}}$$

which, given a starting value $\sigma_{p_{initial}}$, and some constant α, $0 < \alpha < 1$, approximates the geometric mean of the modulus of difference of the observations and the mean to date. The constant α is chosen on the basis of the stability of the observations.

Each individual agent/user pair maintains estimates for the three parameters: *time*, *cost* and *likelihood of success* for the execution of all of its plans, sub-plans and activities. "All things being equal" these parameters are assumed to be normally distributed—the case when "all things are *not* equal" is considered below. *Time* is the total time taken to termination. *Cost* is the actual cost of the resources allocated; for example, time used. The *likelihood of success* observations are binary—ie. "success" or "fail"—and so the likelihood of success parameter is binomially distributed, which is approximately normally distributed under the standard conditions. Unfortunately, *value* is very difficult to measure in process management. The system does not attempt to measure *value*; each individual represents the perceived *value* of each other individual's work as a constant for that individual. Finally, the *delegate* parameter estimates the amount of work delegated to each individual in each discrete time period. The *delegate* parameter is not normally distributed. The *delegate* and *value* estimates are associated with individuals. The *time*, *cost* and *likelihood of success* estimates are attached to plans and activities.

The three parameters *time*, *cost* and *likelihood of success* are assumed to be normally distributed. If working conditions are reasonably stable then this assumption is acceptable, but the presence of external environmental influences may invalidate it. One virtue of the assumption of normality is that it provides a statistical basis on which to query unexpected observations. If an observation lies outside the expected confidence interval then there are grounds, to the chosen degree of certainty, to ask why it is outside. Inferred reasons Γ for *why* an observation is outside expected limits may sometimes be extracted from observing the interactions with the users and other agents involved. If the effect of such a reason can be quantified—perhaps by simply asking a user—then the perturbed values of $\{ob_i\}$ are corrected to $\{ob_i \mid \Gamma\}$.

Delegation may involve forming a group (eg. a committee). Estimating the effectiveness of every possible group of individuals in every possible situation and maintaining the currency of those estimates is not feasible. To deal with groups, the effectiveness of individuals at forming and managing groups is estimated; this is feasible. In this way, to form a group an individual is selected to whom responsibility of forming a group is delegated. In the prototype system, selection may be handled either manually by the user or automatically by the system.

Performance knowledge is historic. If it is used to support future decisions then some allowance should be made for how those performance estimates are expected to have changed in time. For example, if A was good yesterday and B was bad six months ago then how should we rate their expected relative performance tomorrow? The probability of A being better than B will be greater than 0.5. The standard deviation of a parameter can be interpreted as a measure of lack of confidence in its mean. It may be shown that if ρ is the expected range of values for A and B, and if $\sigma_B = \rho$ then the probability of A being better than B will be less than 0.79 no matter what μ_B, μ_A and σ_A are. If $\sigma_B = 2 \times \rho$ then this probability is less than 0.66. So to allow for the historic B estimate, determine a period by which the estimates should be "moderately useless", say one year, and increase σ_B linearly by a half of the difference between its value and $2 \times \rho$ (because six months is half of one year). This has the effect of giving B the "benefit of the doubt" as B has not been given an opportunity for six months.

In the absence of a satisfactory meaning of "best" and with only the performance knowledge to guide the decisions, the approach taken to plan/activity selection is to ask the user to provide a utility function defined in terms of the performance parameters described below. If this utility function is a combination of (assumed) normal parameters then a reasonable plan/activity selection strategy is given a goal to choose each plan (or activity) from the available plans (activities) with the probability that that plan (activity) has the highest expected utility value. Using this strategy even poor plans have a chance of being selected, and, maybe, performing better than expected.

Contract nets with focussed addressing (Ch. 3 by Durfee in (Weiss, 1999)) are used to manage semi-manual or automatic delegation. A bid consists of the five pairs of real numbers (Constraint, Delegate, Success, Cost, Time). The pair *constraint* is an estimate of the earliest time that the individual could address the task—ie. ignoring other non-urgent things to be done, and an estimate of the time that the individual would normally address the task if it "took its place in the in-tray". The Constraint estimates require reference to the user's diary; diary management in the existing system is very basic; (Wobcke and Sichanie, 2000) describes an approach. The pair Delegate is delegations "in" and delegations "out". The pairs Success, Cost and Time are estimates of the mean and standard deviation of the corresponding parameters as described below. The receiving agent then:

- attaches a subjective view of the *value* of the bidding individual;
- assesses the extent to which a bid should be downgraded—or not considered at all—because it violates process constraints, and

- selects an acceptable bid, if any, possibly by applying its 'delegation strategy'.

If there are no acceptable bids then the receiving agent "thinks again".

Given a sub-process, suppose that we have some expectation of the payoff D_i as a result of choosing the i'th individual (ie. agent and user pair) from the set of candidates $\{X_1,...,X_i,...,X_n\}$ to take responsibility for it. A *delegation strategy* at time τ is specified as $S = \{P_1,...,P_i,...,P_n\}$ where P_i is the probability of delegating responsibility at time τ for a given task to individual X_i chosen from $\{X_1,...,X_i,...,X_n\}$. For example, the delegation strategy *best* maximises expected payoff:

$P_i =$

$\begin{cases} 1/m & \textit{if } X_i \text{ is such that } Pr(X_i \text{ »}) \text{ is maximal} \\ 0 & \textit{otherwise} \end{cases}$

where $Pr(X_i$ »$)$ means "the probability that X_i will have the highest payoff" and *m* is such that there are *m* individuals for whom $Pr(X_i$ »$)$ is maximal. Another strategy *prob* also favours high payoff but gives all individuals a chance, sooner or later, and is defined by $P_i = Pr(X_i$ »$)$. An *admissible* delegation strategy has the properties:

- *if* $Pr(X_i$ »$) > Pr(X_j$ »$)$ *then* $P_i > P_j$
- *if* $Pr(X_i$ »$) = Pr(X_j$ »$)$ *then* $P_i = P_j$
- $P_i > 0$ ($\forall i$)

So the strategy *best* is not admissible. The strategy *prob* is admissible and is used in the existing system. It provides a balance between favouring individuals who perform well with giving occasional opportunities to poor performers to improve their performance. The strategy *prob* is *not* based on any model of user improvement and so it can *not* be claimed to be optimal in that sense.

Assessment. The goal-driven system is a distributed multiagent system. This enables the management of complex tasks to be handled as each node is individually responsible for the way in which it goes about its business. That is, the plan in each agent only has to deal with the goals that that agent has to achieve.

The business of delegation of responsibility was discussed above. An over-riding principle is required to determine how delegation is to be dealt with no matter how the measurements as described above are used to support delegation. For example, if A delegates the responsibility for a sub-process to B who, in turn, delegates the same sub-process to C then should B advise A of this second delegation—so removing B from the responsibility chain—or should B remain in the responsibility chain?

The goal-driven system was considerably more expensive to build that the activity driven system. In approximate terms it required four times the programming effort despite the fact that it benefited from being the second system built. Having made this investment dividends flow from the comparative ease by which new processes are included, in that only those agents involved in a process need to develop plans to cope with that process. There is also a negative here. The system has grown around a principle of customisation—ie. each individual is responsible for deciding how their node operates. This means that plans may be constructed at a number of nodes by the users at those nodes to deal with the same sub-process. One way around this is to publish solutions as they are constructed, but that has not been considered.

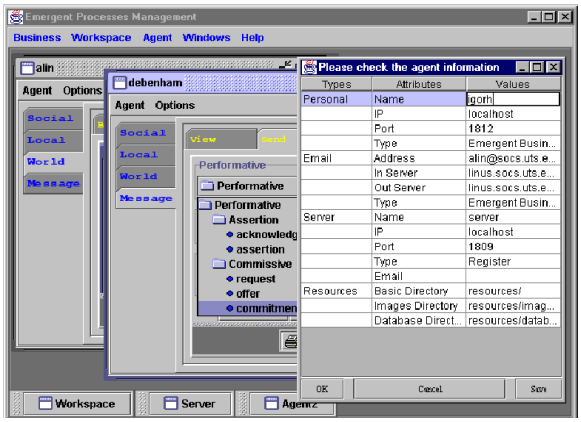

Fig. 6. Setting up a plan in the system

CONCLUSION

Three categories of business process are defined in terms of their management properties. These three categories cover the spectrum from production workflow to emergent process. Two systems have been described that manage activity-driven and goal-driven processes. This sequence of systems is of increasing power and increasing cost to build. The "truth" about process management is a combination of all of these ideas. For example, process knowledge can be important in production workflow—anecdotal knowledge about a "good way to check a proforma" could be valuable process knowledge. The majority of workflow management systems do not capture this sort of knowledge. A powerful process management system should address: the management of goal-driven processes (including activity-driven processes), the support of knowledge-driven processes, the knowledge management aspects of all processes, and the provision of CSCW support that is fully integrated. In the two systems described here, the LiveNet workspace system (Hawryszkiewycz, 1999) is used to handle virtual discussions. Process management requires a solution to the selection problem and the delegation problem. This is achieved on the basis of historical data of past performance and on the basis of inferred reasons for observed deviations in that

performance. Both systems have been implemented in Java. The activity-driven system is implemented as an interpreter of statecharts annotated with event-condition-action rules. The goal-directed system is implemented as an interpreter of high-level agent specifications. This interpreter enables agents to be built quickly. It also simplifies maintenance, which only has to deal with high level specifications of goals and plans. Both systems use virtual documents, and all three systems may be used remotely. All that is required is an internet connection. Fig. 6 shows the screen for entering a plan into an agent's plan library in the goal-driven system.

REFERENCES

Debenham, J.K. (2000). "Supporting Strategic Process", in proceedings Fifth International Conference on The Practical Application of Intelligent Agents and Multi-Agents PAAM2000, Manchester UK, April 2000.

Dourish, P. (1998) "Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications." ACM Transactions on Computer-Human Interaction, Vol. 5, No. 2, June, 1998, pp. 109—155.

Finin, F. Labrou, Y., and Mayfield, J. (1997). "KQML as an agent communication language." In Jeff Bradshaw (Ed.) Software Agents. MIT Press (1997).

Fischer, L. (Ed) (2000). "Workflow Handbook 2001." Workflow Management Coalition & Future Strategies, 2000.

Hawryszkiewycz, I.T. (1999). "Supporting Teams in Virtual Organisations." In Proceedings Tenth International Conference, DEXA'99, Florence, September 1999.

Jennings, N.R., Faratin, P., Norman, T.J., O'Brien, P. and Odgers, B. (2000) "Autonomous Agents for Business Process Management", Int. Journal of Applied Artificial Intelligence 14 (2) 145—189.

Müller, J.P. (1996). "The Design of Intelligent Agents" Springer-Verlag.

Muth, P., Wodtke, D., Weissenfels, J., Kotz D.A. and Weikum, G. (1998). "From Centralized Workflow Specification to Distributed Workflow Execution." In Journal of Intelligent Information Systems (JIIS), Kluwer Academic Publishers, Vol. 10, No. 2, 1998.

Rao, A.S. and Georgeff, M.P. (1995). "BDI Agents: From Theory to Practice", in proceedings First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, USA, pp 312—319.

Singh, M.P. and Huhns, M.N. (1999). "Multiagent Systems for Workflow", International Journal of Intelligent Systems in Accounting, Finance and Management, Vol 8, 1999, pages 105—117.

Weiss, G. (Ed) (1999). "Multi-Agent Systems". The MIT Press, Cambridge, MA.

Wobcke, W. and Sichanie, A. (2000). "Personal Diary Management with Fuzzy Preferences" in proceedings Fifth International Conference on The Practical Application of Intelligent Agents and Multi-Agents PAAM2000, Manchester UK, April 2000.