RECOVERY SERVICES FOR THE PLANNING LAYER OF AGENTS

Khaled Nagi, George Beskales
Computer and Systems Engineering Department,Faculty of Engineering, Alexandria University, Egypt.

Keywords: Software agents, planning layer, recovery services, agent robustness, agent simulation.

Abstract: Software agents represent aim at automating user tasks. A central task of an agent is planning to achieve its goals. Unexpected disturbances occurring in the agent execution environment represent a serious challenge for agent planning. In this work, a recovery model for the planning process of agents is proposed to cope with disturbances caused by system failures; which often lead to system crashes. The proposed recovery model supports the Hierarchical Task Networks (HTN) planners which represent a broad family of planners that are widely used in agent systems. A prototype for the proposed recovery services is implemented to demonstrate the feasibility of the proposed approach. Furthermore, a simulation is built and many simulation experiments were conducted to gain insight about the performance of the proposed recovery model.

1 INTRODUCTION

Software agent systems are introduced to ease automatable tasks of the end user. One of the important features that are required to exist in the developed agents is *robustness*. Relying on agents to perform critical tasks necessitates that they possess a high level of reliability to overcome possible failures in any of the agent components or in its environment.

Various agent architectures tend to group agent components into subsystems, i.e. *layers*. Typical layers found in existing agent systems can be classified into the following:

Cooperation Layer: this layer is responsible for the social ability of the agent. Interaction and communication with other agents in the environment are handled by this layer in order to allow goal sharing and plan execution sharing.

Planning Layer: having set a goal to achieve, the agent begins to generate a plan -which can be considered as a set of actions- to reach that goal. The agent accepts the current state of the environment, the goal to achieve and the possible actions allowed in the environment. The agent then attempts to generate a plan that accomplishes these goals.

Execution Layer: This layer is responsible for executing the planned actions in the agent environment. This layer is also responsible for scheduling the tasks for execution to avoid invalid tasks interleaving. Execution layer may include interface components that are required to submit the agent actions to the environment, e.g. database interfaces.

1.1 Robustness of Agents

Robustness of agents has long been considered by agent system developers to provide a reliable execution of agents. A robust agent can be defined as an agent that is capable to identify and overcome a finite set of failures in order to allow transparent continuation of agent operation.

Robustness should cover all layers of the agent to handle all possible failures/disturbances at each layer. Most of previous work in this context concentrated on the cooperation layer and the execution layer. Our recovery model addresses the planning layer to handle disturbances due to system failures; either hardware or software. Typical types of disturbances are *the crash of the agent,* and hence the loss of its generated plans, or *unplanned changes of the state of the agent environment,* which may make the generated plan invalid. We assume the correctness of the planning algorithms and hence the validity of the generated tasks. Therefore, our solution does not handle crashes due to logical errors in the plan itself.

Naturally, after a disturbance occurs, the agent has to restart the planning process from scratch if no recovery services are provided. Recovering of the generated plan minimizes the penalty of these disturbances by restoring a valid plan and returning it to the planner.

Nagi K. and Beskales G. (2006).
RECOVERY SERVICES FOR THE PLANNING LAYER OF AGENTS.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - SAIC*, pages 138-146
DOI: 10.5220/0002443601380146
Copyright © SciTePress

2 RECOVERY MODEL

A recovery model is built to provide a robust planning process of agent systems. A lot of planning algorithms can be exploited to perform the planning of the agent goals. One family of these planners is the Hierarchical Task Networks (HTN) (Erol, et al., 1994) and (Erol, 1995). The planners under this family achieve a plan for a set of goal tasks by decomposing them into more primitive tasks. The decomposition process is repeated until the most primitive tasks are achieved. There are many implementations for HTN such as NOAH, NONLIN, MOLGEN, UMCP, SHOP, SHOP2 (Nau et al., 2003) and others. This family of planners is adopted to be supported by the proposed recovery service to prove the concept of plan recovery.

2.1 Overview of HTN Planning

HTN planners accept the initial state of the environment, the goal tasks, and the definition of the problem domain as input. The output of the planner is a plan that can achieve the goal tasks.

Tasks are classified to primitive and non-primitive tasks. Non-primitive tasks are abstract tasks that must be decomposed by the planner to a set of more primitive tasks. Primitive tasks are the tasks that can be achieved by a single action (operator).

An environment state *S* is a set of ground atoms that are true in the environment. The initial state at the beginning of planning is denoted as S_0. An operator α is an action that can be executed in the agent environment. It can be represented by a tuple *(head, parameters, precond, del, add)*, where *head* is the operator name, *parameters* are its variable parameters, *precond* is a list of precondition that must be true in the environment state before execution of the operator, *del* is a list of terms to be removed from the state after execution, and *add* is a list of terms to be added to the state.

Define a function named *RESULT* that maps a pair of a state and an operator to another state:
`RESULT: S × Op → S`

A plan *P* is a set of operators with a dependency between them. Formally, a plan $P= \{\{(n_1{:}\alpha_1),\ldots,(n_k{:}\alpha_k)\},<_p\}$, where α_i is an operator in the plan with labels n_i to distinguish similar operators. Define a function *PL_RESULT* that maps an initial state and a plan to a final state: `PL_RESULT: S × P → S`. *PL_RESULT* applies the operators in plan *P* successively in their precedence order against an initial state S_0 to achieve a final goal state S_g. A method *m* is used to decompose a non-primitive task. A method *m* can be defined as follows:

$m = (head, parameters, precond, subtasks, <_m)$, where *head* is the method name, *parameters* are a list of method parameters, *precond* are the preconditions of the method, *subtasks* are a set tasks that can accomplish the goal task, and $<_m$ is the dependency relation between tasks in *subtasks*. A problem domain *D* is defined as a tuple *(At,Op,Me)* , where *At* is a list of all possible atoms in this domain, *Op* is a list of possible operators and *Me* is a list of possible methods. A planning problem $Pr= (D,S_0,T)$ consists of a problem domain *D*, an initial state S_0 and goal tasks *T*. Its solution is a plan *P* that can achieve the goal tasks.

Generally, an HTN planner, e.g. SHOP2, repeatedly picks (and removes) some task from *T*, decomposes it to more primitive tasks and returns them to *T* until the most primitive tasks remain in *T*. At this point, each primitive task can be achieved by a single operator. The operators used are appended to the output plan and the plan is then returned to the execution layer after all tasks are achieved.

2.2 Assumptions

We assume that the agent environment is *open*, i.e. it can be modified by other external entities, and *history-independent*, i.e. the outcome of an operator does not depend on the previously executed operators. Also, it is assumed that the execution process of the agent checks the satisfaction of all operator preconditions before executing.

Plan execution can begin before the completion of its generation. In other words, generated tasks of a partially developed plan can be executed by the agent. This feature is especially important in agent systems where planning can take considerable time, which motivates the necessity of recovery. Some planners cannot allow early execution of plans for certain reasons such as the use of backtracking algorithms where a generated task be removed at a late planning step. We adopt the more general case where early execution of partially developed plans is allowed.

2.3 Architecture of the Proposed Recovery Model

Figure 1 depicts the architecture of the proposed recovery model. A typical agent system is used as a basis for the recovery service. The agent system consists of three typical layers: *Execution Process*, *Planner*, and *Cooperation Process*. The plan recovery service is tightly coupled with the planning process to provide recoverable plans. The planner initially submits the planning problem which is represented by the tuple *(D,S_0,T)*. Every time the planner generates a new task, it has to submit it to the recovery service so that the persistent image of the plan can be updated. Through its interaction with the execution process, the planner communicates the schedule and the state of execution of the plan to the recovery service module. Upon disturbance, the recovery service module recovers a valid plan with respect to the new environment state and returns it to the planner so that it can continue planning. The recovery service uses a persistent storage unit to store an image of the submitted planning information so that it can be retrieved after a crash occurs.

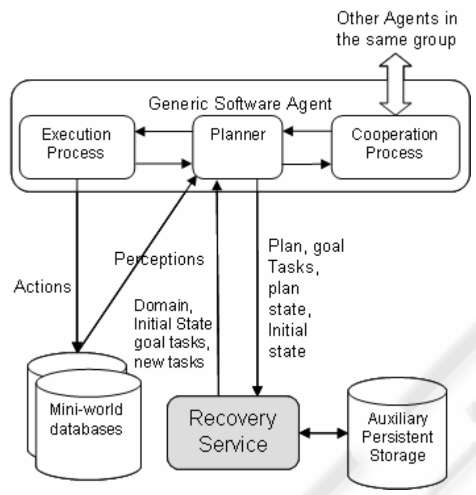
Other Agents in the same group
Generic Software Agent
Execution Process
Planner
Cooperation Process
Perceptions
Actions
Domain, Initial State goal tasks, new tasks
Plan, goal Tasks, plan state, Initial state
Mini-world databases
Recovery Service
Auxiliary Persistent Storage

Figure 1: Agent system architecture with integration of planning recovery service.

The traditional data structures used in the HTN planner, and which are also used in the recovery module, are the plan *P*, the goal tasks *T*, the initial state S_0, the planning state S_p and the problem domain *D*. Additionally, we introduce a new data structure, the *Plan Task List* (*PTL*), to enable successful recovery of plans. These data structures represent the state of the planner at any point of time and hence they have to be stored and maintained by the recovery module. The *PTL* data structure preserves the following:

– tasks hierarchy, i.e. the parent-child relation between generated tasks,
– operators used to achieve primitive tasks,
– methods used to decompose non-primitive tasks,
– tasks that are already achieved or decomposed which are discarded from the goal tasks *T*.

Formally the *PTL* can be represented as follows:

PTL=$\{(t_i, \text{subtasks}_i , m_i, \alpha_i , \theta_i), <_p\}$, $0 \le i \le$ no of nodes in the tasks hierarchy; where $subtasks_i$ are the subgoals of t_i, α_i is the operator to achieve t_i, m_i is the method used to decompose t_i to $subtasks_i$, and θ_i is the used binding that unifies the operator or the method with the task. $Subtasks_i$ and m_i must be equal to ϕ whenever t_i is primitive, and α_i must be ϕ if t_i is non-primitive. Also, $subtasks_i$, m_i, α_i and θ_i must be equal to ϕ if $t_i \in T$ because at this point the task t_i is not yet achieved by an operator, in case of primitive task, or decomposed, in case of non-primitive task. Each task in the goal tasks *T* is initially copied to *PTL*. When a task is decomposed, its subgoals are associated with it without deleting it from *PTL*.

In (Beskales, 2005), we prove that this set of data structures is necessary and sufficient for a correct recovery of plans according to procedures mentioned in the next section.

2.4 Logging and Recovery Procedures

In order to allow the recovery service to do its function, logging procedures must be implemented to record newly generated tasks by the planner and to update the *PTL* structure accordingly. These procedures are: *Initialize*, *SubmitPrimitiveTask*, *SubmitNonPrimitiveTask* and *UpdateTaskExecStatus*. Due to space limitation, a pseudo code for each of these procedures is omitted. They can be found in (Beskales, 2005). The procedure *Initialize* must be invoked at beginning of planning. It stores new planning problems in the persistent storage and initializes the *PTL* structure. The procedures *SubmitPrimitiveTask* and *SubmitNonPrimitiveTask* must be invoked by the planner after a primitive task is achieved by an operator and a non-primitive task is decomposed using a method, respectively. They store the subtasks and the used method in case of a non-primitive task, or the used operator in case of a primitive task. The *PTL* and *P* structure are updated accordingly. All modified structures are updated in the persistent storage. The procedure *UpdateTaskExecStatus* should be invoked by the execution process when a task is executed or by the planner if it has a full knowledge about the execution status of tasks from the execution process. The procedure *RecoverPlan*, listed below, is invoked by the planner after occurrence of disturbances. It retrieves the stored planning data from the persistent storage and starts to evaluate the validity of each item with respect to the current environment state. Only valid plan operators are returned in the plan while invalidated tasks are removed from the plan and are added to the goal tasks for replanning. The returned data represents a consistent state for the planner, from which it continues processing.

```
**recoverPlan**(State $S_c$): P, T, $S_0$, $S_p${
/*invoked after a disturbance. New environment state is $S_c$ */
if(crash occurred)
  restore(D,P,T,PTL,$S_0$,$S_p$);
$S_c$'=$S_0$;   // $S_c$' is the supposed state
iterate over t∈P in prec. order $<_p${
  if(t.execStatus = 'completed'){
    $S_c$'= RESULT($S_c$', t.$\alpha$);
    removeTask t from PTL and P;
  }
}
$S_0$=$S_c$; //update $S_0$ to the current env. state
if($S_c$==$S_c$'){
  return P,T,$S_0$,$S_p$;
}else {       //$S_c \neq S_c$'
  $S_p$=$S_c$;
  iterate over t∈PTL in depth first traversing in precedence order ($<_p$){
    if((t == primitive $\wedge$ t.$\alpha$.precond
       is not satisfied by $S_p$) $\vee$ (t is
       not primitive $\wedge$ t.m.precond is
       not satisfied by $S_p$)){
      Task[] SubTasks = $\phi$;
      Task[] Dependents = {t};
      Loop until SubTasks and Dependents
        stabilize{
         SubTasks = SubTasks  $\cup$
           {t'∈ PTL: ancestor(t'',t') $\wedge$
           t''∈ Dependents $\cup$ SubTasks};
         Dependents = Dependents $\cup$
           {t'∈PTL: t'' $<_p$ t' $\wedge$
           t''∈ SubTasks $\cup$ Dependents};
      }
      Dependents = Dependents-SubTasks;
      For each task $t_s$∈ SubTasks{
        PTL.removeTask($t_s$);
        T.removeTask($t_s$);
        PTL.removeBinding($t_s$.$\theta$);
        T.removeBinding($t_s$.$\theta$);
        if($t_s$ is primitive $\wedge$ $t_s$∈ P)
           P.removeTask($t_s$);
      }
    For each task $t_p$∈ Dependents{
      PTL.removeBinding($t_p$.$\theta$);
      T.removeBinding($t_p$.$\theta$):
      T.addTask($t_p$);
      Task t'=PLT.searchForTask($t_p$)
      t'.operator = t'.method =
      t'.bindings = t'.subtasks = $\phi$;
      if($t_p$ is primitive $\wedge$ $t_p$∈ P)
        P.removeTask($t_p$);
    }
  }else  // t.precond is satisfied by $S_p$
    $S_p$ = RESULT($S_p$,t.$\alpha$);
  }
}
update (P,T,PTL,$S_0$,$S_p$);
return P,T,$S_0$,$S_p$;
```

An example of plan recovery is shown in Figure 2. After generating a plan $P=(t_0, t_1, t_2, t_3, t_4)$ with remaining goal tasks $T=\{t_5,t_6,\ldots\}$, and assuming that t_0 and t_1 are executed, a disturbance occurs. Invocation of the recovery procedure will result in the following:

– Removing all completed tasks, i.e. t_0 and t_1.
– Traversing the non-completed tasks in P and checking their validity with respect to the new environment state. This results in removing invalid tasks t_4.
– Removed tasks are added to the goal tasks T to be replanned.
– Finally, new initial state S_0 (which equals the current environment state S_c) and new planning state S_p are calculated and returned to the planner along with the modified P and T.

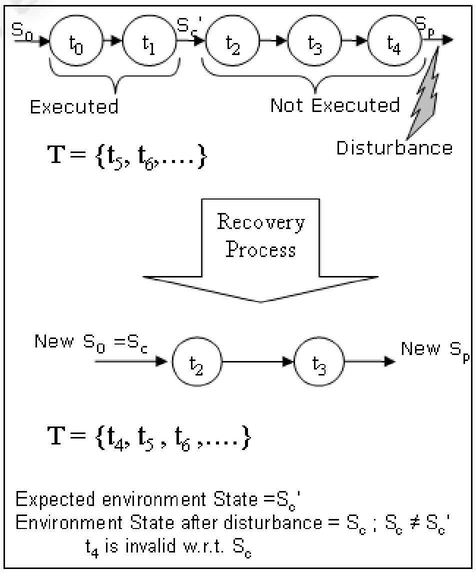
S_0
t_0
t_1
S_c'
t_2
t_3
t_4
S_p
Executed
Not Executed
Disturbance
$T = \{t_5, t_6, \ldots.\}$
Recovery Process
New S_0 =S_c
t_2
t_3
New S_p
$T = \{t_4, t_5, t_6, \ldots.\}$
Expected environment State =S_c'
Environment State after disturbance = S_c ; $S_c \neq S_c$'
t_4 is invalid w.r.t. S_c

Figure 2: An example of plan recovery.

3 SIMULATION MODEL

3.1 The Prototype

In order to verify the feasibility of the proposed recovery model, a prototype is built that utilizes the logging and recovery procedures described earlier. The modules of the prototype, illustrated in Figure 3, are described below.

We chose the *JSHOP2 planner* (Ilghami) to be integrated with our recovery service. JSHOP2 is an open-source Java implementation for the SHOP2 planner, which takes the problem domain and goals as input parameters. The *Recovery Module* is responsible for storing and recovering the generated plans. It mainly consists of the implementation of the logging procedures and recovery procedure; described in Section 2.4. The *Simulated Persistent Storage* is a transient data structure used to keep an image of the recovery information submitted by the recovery service. In our simulated environment, there is no need to actually store the plan in a physical persistent storage because the crashes are simulated by another module rather than actually performed. Upon receiving a crash signal, the planner internal variables are nullified, the recovery procedure retrieves the stored recovery information from the simulated persistent storage, and finally a valid plan is returned to the planner after removing invalid tasks. The *Crash/State Change Simulator Module* is used to generate disturbances including simulated crashes of the planner and unplanned changes of the environment state according to the probability distributions mentioned in Section 3.2. The *Planning/Recovery Monitor* is responsible for observing the output of the planner as well as the recovered plans returned by the recovery service after a disturbance. By observing system outputs, the user can validate the correctness of the planner and the recovery service and gather the necessary measurements for the performance metrics described in Section 3.3.

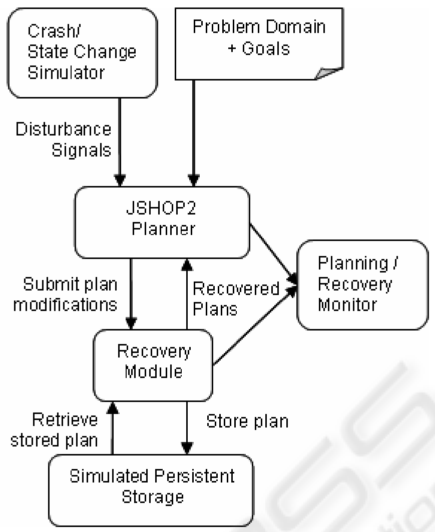

Figure 3: The implemented prototype.

3.2 Probability Distributions Used in the Simulation

The inter-arrival time between disturbances (in terms of processed tasks) follows an exponential distribution with mean T_{dst}. The probability of change of each ground atom in the environment state follows a Bernoulli distribution with probability of modification P_{mod}. If a ground atom is selected for modification, the new atom is randomly selected from all possible states. The number of completed operators of the generated plan at time of disturbance is uniformly distributed in the range from zero to the length of the partially generated plan. The probability of crash occurrence given that a disturbance already occurred is assumed to follow a Bernoulli distribution with expected probability of crash occurrence equal to P_{cr}.

3.3 Performance Metrics

The following performance metrics are used in the simulation study:

– *The number of processed nodes used in the plan generation is the main performance metric*. In the presence of recovery, the total number of processed tasks is calculated including those processed by the planner and the recovery modules. The ratio of the number of tasks processed by the recovery procedure to the number of tasks processed by the planner is also monitored to provide an indicator about the relative overhead of the recovery service with respect to the planner.

- *The number of performed I/O operations* is also monitored. To count the I/O operations, the number of read/written bytes is monitored in each of the I/O events and then the number of I/O accesses is calculated at each event by dividing the number of transferred bytes over the number of bytes per page.
- The storage requirements of recovery are also recorded to obtain a complete image about the requirements of the recovery process. The storage requirement is defined as the maximum number of bytes stored in the persistent storage for the duration of the planning process.

3.4 Recovery Trade-off

Recovery service introduces new overheads as a result of logging of planned tasks and executing recovery procedure after disturbances. These overheads can be justified if the recovery service can actually assist the planner to retract to a mature planning state rather that restarting the planning process from scratch.

The proposed logging procedures and recovery procedure have the following overhead:

- Fixed I/O overhead: due to logging of planning events.
- Storage overhead: used to store the logged planning information.
- Recovery overhead: encountered when recovery procedure is invoked after occurrence of disturbance. It is composed of additional I/O operations to retrieve the stored image of *PTL*, *T* and *P* from the persistent storage in case of crash, and the number of processed tasks in order to check the validity of each task in *PTL* before returning the recovered plan and goal tasks to the planner.

The cost of the planning process consists of:

- Processed tasks: in order to generate a plan for the goal tasks, i.e. to decompose them using domain methods and operators to achieve a plan, and
- I/O operations: which are performed initially and after each disturbance to read current environment state, i.e. perceptions.

For the recovery approach to be efficient, the sum of the planning cost and logging/recovery costs must be less that the planning cost in absence of recovery services. In such case, the source of recovery efficiency is that it allows the planner to retract to a mature, consistent and valid planner state rather than reinitializing its state after occurrence of disturbances. The number of disturbances must overpass a specific threshold and the recovered planner state should be as mature as possible. To find out these thresholds, several experiments are conducted based on the prototype under various conditions.

4 EXPERIMENT RESULTS

4.1 Effect of Varying T_{dst} and P_{mod}

As expected, there is a significant performance gain when the recovery procedure is used especially at high disturbance rates, i.e. low values of T_{dst}, and in semi-static environments, i.e. at low values of P_{mod} (P_{mod}= 0.05) as illustrated in Figure 4 and 5. In absence of recovery services, the planner has to restart with an empty plan each time after disturbance occurrence while the recovery process restarts the planner from a more mature state. It is normal that increasing rate of disturbances, i.e. decreasing T_{dst}, leads to increasing the gap between the two approaches. This is mainly because each additional disturbance costs the planner further penalty when the recovery service is disabled compared to the case where the recovery service is exploited.

In Figure 5, it is noticed that the number of performed I/O operations at high values of T_{dst}, is slightly higher when the recovery service is used. As T_{dst} begins to decrease, the recovery enabled planning performs better than the traditional approach in terms of I/O operations. This observation is expected because at low rate of disturbances, the recovery service performs a greater number of I/O operations to log planned tasks, compared to the traditional approach where I/O operations are only performed for perceptions. At higher rates of disturbances, the duration of planning time when recovery services are disabled is much greater than the duration when recovery service is used because each disturbance has much penalty. As a result, the number of disturbances in the complete duration of planning is much greater when no recovery is used. This is directly reflected on the increasing number of I/O operations performed to read the environment perceptions after each disturbance, which exceeds the logging and recovery I/O overhead.

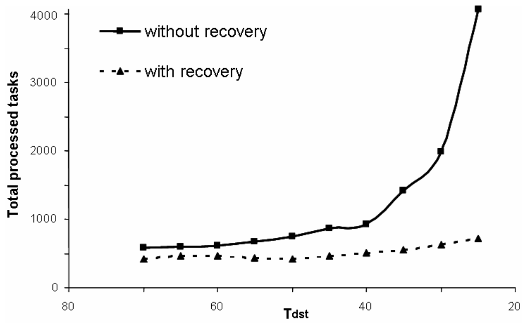

Figure 4: Total processed tasks (P_{mod} = 0.05).

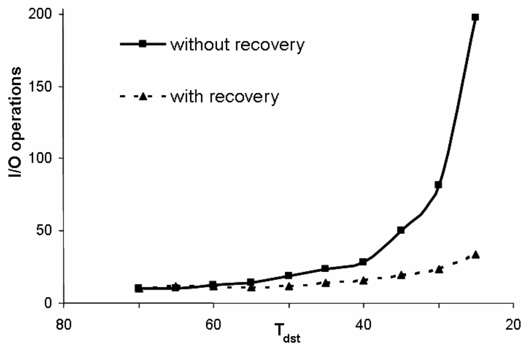

Figure 5: Performed I/O operations (P_{mod} = 0.05).

Figures 6 and 7 depict the processed tasks and I/O operations in a highly-dynamic environment (P_{mod}=0.95) where environment state dramatically changes after disturbances. It is observed that the recovery service is inefficient at highly dynamic environments regardless of the rate of disturbances because the function of recovery procedure is most likely useless.

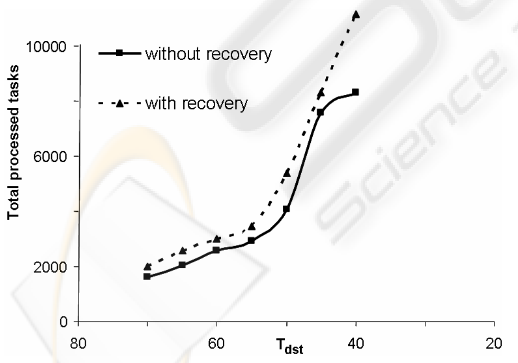

Figure 6: Total processed tasks (P_{mod} = 0.95).

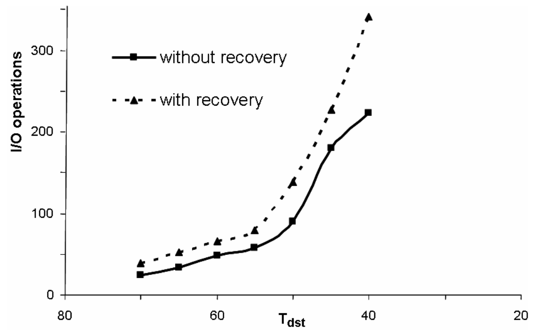

Figure 7: Performed I/O operations (P_{mod} = 0.95).

The main reason is that at very high values of P_{mod}, the recovery service loses its advantage to recover to a mature state of the planner and degenerate to the same effect of the traditional approach, which leaves the logging I/Os as an unjustified overhead.

In Figure 8 and 9, we fix T_{dst} at 80 and vary P_{mod} from 0.1 to 0.9. We observe different breakeven points than with the previous sets of experiments. Here the value of P_{mod} should not exceed 0.45 in order to make the recovery enabled approach cost-effective in terms of both CPU and I/O operations.

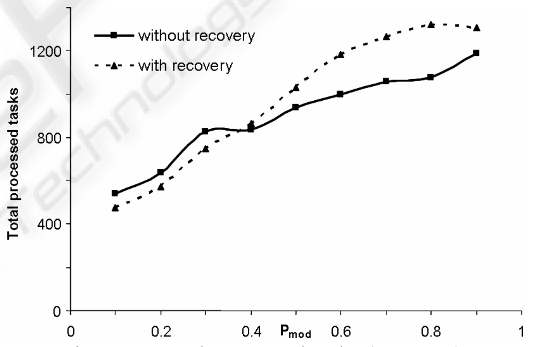

Figure 8: Total processed tasks (T_{dst} = 80).

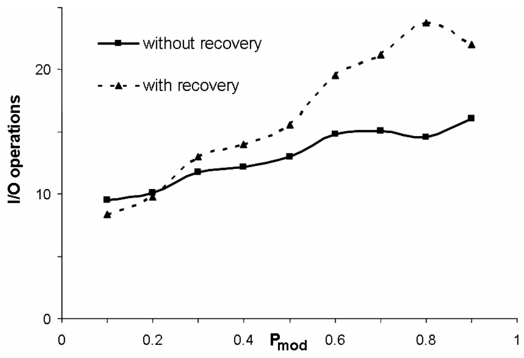

Figure 9: Performed I/O operations (T_{dst} = 80).

So, we have to find out the constraints for efficient plan recovery. In other words, we have to define the environment in which the recovery-based planning supersedes the traditional non-recoverable planning. In order to determine the thresholds of

P_{mod} for other values of T_{dst}, we executed multiple experiments for all values of P_{mod} and T_{dst}. The threshold values can be found in Figure 10.

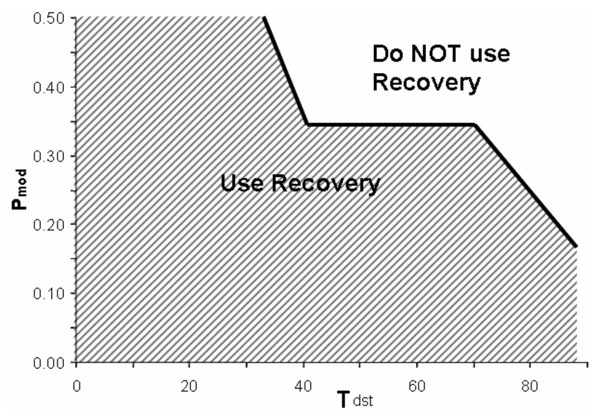

Figure 10: Constraints for efficient use of recovery.

The area below the curve contains the values of T_{dst} and P_{mod} that make the recovery-based approach effective. A conservative approach is adopted while identifying the threshold values so that *both* the number of processed tasks and I/O operations of our approach must be lower than those of the traditional approach to conclude that the former is more efficient.

Since the absolute value for this threshold line varies with the problem domain, the simulator used for analyzing the performance of the proposed approach, can also be used as a system management tool that helps system administrators to keep performance requirements. Thresholds for T_{dst} and P_{mod} can be obtained for the specific problems by the simulator. Then, the values of T_{dst} and P_{mod} of the agent environment can be monitored at the system operation, and a decision for enabling/disabling the recovery services can be taken accordingly.

4.2 Storage Requirements

Figure 11 depicts the storage overhead against varying values of T_{dst} in a semi-static environment where P_{mod}=0.05. It is observed that increasing the rate of disturbances results in less storage requirements. This is because increasing rate of disturbances reduces the probability of developing one complete image of plan which means less expected storage requirements.

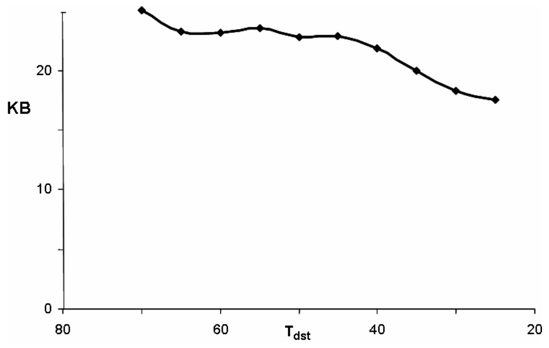

Figure 11: Storage overhead (P_{mod} = 0.05).

According to Figure 12, it is noticed that increasing P_{mod} results in insignificant changes to the storage requirements, mainly slightly increasing values, due to increasing probability of state change and increasing overall planning time which in turn generates longer plans to cover all encountered states, especially those far from the goal states. In general, we can state that the storage requirement is quantitatively small and is relatively insensitive to the change in the environment parameters.

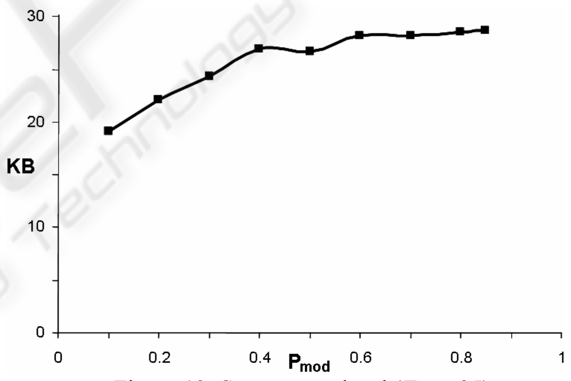

Figure 12: Storage overhead (T_{dst} = 35).

5 CONCLUSION

Due to inherent problems of dynamic and unstable environments of software agents, we propose a recovery model for the central layer of typical agent systems: *the planning layer*. It is intended to support a widely used family of planners known as the Hierarchical Task Network planners. We select SHOP2 as a representative of HTN-based planners as a concrete model to prove the concept.

The proposed recovery module introduces a set of logging procedures to record planner actions on persistent storage. A recovery procedure is proposed to restore the planning information from the persistent storage and to process the retrieved plan and goal tasks. It removes invalid tasks before returning the recovered planning state to the planner.

A simulation model is built over the prototype to monitor the recovery process behaviour under various conditions and to extract the conditions under which the recovery service is efficient. According to the simulation results, the recovery service provides an efficient overall system performance in semi-static environments at high rates of disturbances. This is the expected behaviour of the recovery services as restarting the planner from a mature state is more efficient than restarting from the planning initial state. In opposite cases where agents act within failure-free environments, the recovery-enabled systems will suffer from unjustified logging overhead. Also, a highly dynamic environment state will make the recovery of stored plan inefficient as larger portions of the previously generated plan will be obsolete and will be discarded in such cases.

REFERENCES

Beskales, G., 2005. Recovery services for the Planning Layer of Agents, M.Sc. Computer and Systems Engineering Department, Alexandria University.

Erol, K., Hendler, J., and Nau, D. UMCP, 1994. A Sound and Complete Procedure for Hierarchical Task-Network Planning, Institute for Systems Research, University of Maryland.

Erol, K., 1995. Hierarchical task network planning: Formalization, analysis and implementation. University of Maryland at College Park, Dept. of Computer Science.

Ilghami, O. Documentation for JSHOP2. TR CS-TR-4694 www.cs.umd.edu/projects/shop/

Nau, T., Ilghami, O., Kuter, U., Murdock, U., Wu. D., and Yaman, F. 2003. SHOP2: An HTN planning system. Journal of Artificial Intelligence Research, 20: pp. 379-404.