A FLEXIBLE CONTROL MECHANISM FOR MANAGING INTERRELATED/INTERDEPENDENT TASKS SUCCESSIVELY

Hiroyuki Watanabe
Toyohide Watanabe
Noboru Sugie

Reprinted from PROCEEDINGS OF THE EIGHTEENTH ANNUAL INTERNATIONAL COMPUTER SOFTWARE & APPLICATIONS CONFERENCE (COMPSAC 94)
Taipei, Taiwan, November 9-11, 1994

IEEE Computer Society
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

Washington, DC • Los Alamitos • Brussels • Tokyo

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.
IEEE

IEEE COMPUTER SOCIETY

A Flexible Control Mechanism for Managing Interrelated/Interdependent Tasks Successively

Hiroyuki WATANABE, Toyohide WATANABE and Noboru SUGIE
Department of Information Engineering,
Faculty of Engineering, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-01, JAPAN

Abstract

In this paper, we propose a framework to manage interrelated tasks concurrently and successively, and address the issue about event handling and task scheduling: the event handler organizes interrelated tasks from a static point of view, and the task scheduler controls them on the basis of the dynamic behavior properties. In order to design our flexible control mechanism, we introduce the state transition graph and AND-OR graph in the event handler, whereas in the task scheduler the timestamp ordering method is applied to. Additionally, we describe the failure handling, concerning the cooperation between the event handler and task scheduler.

Introduction

We address a concurrent control mechanism among interrelated tasks on the basis of the concepts of cooperativeness and successiveness. Our final research goal is to construct a secretary model, which can manage many different jobs effectively and flexibly like a well human secretary, and in this paper the basic task control mechanism is investigated as the first step. The skillful human secretary can perform different kinds of jobs intelligently in accordance with requests: if these jobs were interrelated, some related jobs are integrated smartly or synchronized instantly. Thus, our control mechanism should support such an intelligent activity.

The cooperativeness for tasks is frequently argued as the issue about interactive relationships among independent agents [1–3]. These researches divide the control mechanism into several autonomous agents. Such an approach is suitable to the distributed environment. However, it is insufficient to construct a secretary model because a well human secretary can act/react newly generated jobs or temporarily suspended jobs promptly in accordance with her processing situation and environment. Since the distributed model based on multi-agents delegates the control mechanism for problem solving strategy to independent agents, a number of distributed agents can not adapt to variant changes of situation and environment smartly.

The successiveness for tasks may be looked upon as the issue about connective relationships among interdependent transactions [4]. In particular, the researches for managing long-lived transactions focus on effective and consistent control mechanisms. Such a view is very similar to our motivation. However, many approaches control long-lived transactions as the nested structure. The superior transaction must wait for commitment messages sent from all inferior actions(or transactions). At least, such a control mechanism is not adaptable to our objective for making the total throughput shorter. SAGA was proposed as one of the non-nested transaction models [4]. The concept of compensated transaction in SAGA is valid to keep the integrity constraint of database system. However, such a concept makes it difficult to construct a secretary model because many compensated transactions must be defined, corresponding to all transactions.

From viewpoints of cooperativeness and successiveness among interrelated tasks, it is necessary to distinguish individual tasks related to some demands and determine the execution order among them. Additionally, the synchronous control and commit/abort control mechanisms among executed tasks must be implemented successfully with a view to accomplishing the high throughput. With respect to these requirements, our framework for controlling interrelated tasks flexibly is based mainly on the event handling and task scheduling: the event handler must distinguish mutually related tasks derived from some events and establish the interdependent relationships among them. While, the task scheduler must select executable tasks according to the current management state and determine the execution order between the scheduled tasks and newly generated tasks effectively.

Framework

First of all, we define some composite concepts for our model: event, task, action and object, as follows:

- The event is a request to be performed practically as some tasks. An event is not only corresponded to a task but also may generate a collection of tasks;
- The action is derived from the event and generates a collection of related tasks. Namely, the action is an interpreter to translate an event into the corresponding related tasks;
- The task is a self-contained command. In our model, the tasks are defined as messages for objects;
- The object is a task executer. When objects receive tasks, they evaluate the tasks concurrently.

For example, we consider a request of meeting preparation. We show the related tasks in Figure 1. In Figure 1, hexagons indicate events, hatched rectangles represent tasks and arrows control the execution order among

0730-3157/94 $04.00 © 1994 IEEE

them. In Figure 1, the task "Appoint a meeting room" is directly generated from the event "Decide a meeting" and may be executed if possible. The task "Add to roll-book", which associates with this request, must be started after the event "Receive responses". The tasks in Figure 1 are managed by two actions. Also, we represent the correspondence between tasks and objects in Figure 2. In Figure 2, circles indicate objects. The object is either a receiver or argument of the task, and the receiver evaluates the task as a message. These concepts facilitate not only modelling secretarial jobs but also tranforming the request into several related tasks.

In our framework, the event as a request is finally managed by a group of corresponding objects, which are atomic processing actors. In addition to the effectivity of this transformation, task executers are controlled concurrently and synchronously in order to perform the original request correctly. However, in the execution process, it is not always successful to keep the predefined evaluation order of tasks, which were coordinated statically by actions. In some cases, it is desirable that the tasks, which are scheduled as the next ones of currently executing tasks, should be started quickly even if any tasks are not yet completely finished by the corresponding objects. As for this requirement, our framework comprises two components:

- Event Handler: The event handler selects firstly an appropriate action according to the event and determine mutually related tasks through the action. In this case, the execution order is assigned to these related tasks. The tasks are finally entered into the task queue according to the execution order. Namely, the event handler generates tasks from the event through the corresponding action, and also schedules these generated tasks statically;
- Task Scheduler: The tasks should be controlled effectively according to the current state, but not according to the statically planed sequence. Therefore, the task scheduler analyzes the dynamic interaction among tasks and reschedules the interrelated tasks on the basis of the predefined execution order. In addition, the task scheduler manages all behaviors of task executers(i.e. objects). Namely, the task scheduler controls the task execution according to the current situation, and reschedules the execution order among tasks if necessary.

Task Ordering

The event handler selects appropriate actions according to the event and generates the related tasks through the action. In this case, it is important to represent the relationship between events and tasks clearly.

1. State Transition Graph

In general, the evaluation order among events may be predefined. For example, if a human secretary sends an invitation letter, she expects that the event "Receive a response of the invitation" should be occurred in the future. Our state transition graph represents the evaluation order among events and the relationship between events and actions. We show an example of our state transition graph in Figure 3. Figure 3 represents a job for organizing a meeting. Here, $Si(i = 1, ...)$ is a state and the arrow shows the correspondence between event and action. Namely, the left(or upper) argument of arrow is an event and the right(or lower) argument is the corresponding action. In Figure 3, when the event "Decide a meeting" occurs, the state changes to $S1$ and the action "Send invitation letters" is fired.

A state transition graph indicates a self-contained job conceptually and represents the transition of activities. For example, the state transition graph in Figure 3 represents a job for preparing a meeting and indicates which actions are fired, corresponding to successively requested events. The event handler has several state transition graphs to represent currently executing activities. When the event handler receives an event, it decides which state transition graph can accept the event, and also changes the state or creates a new state transition graph if the event indicates a new activity.

2. AND-OR Graph

AND-OR graph represents the relationships between fired actions and related tasks, and indicates the execution order among these tasks. We show an example of AND-OR graph in Figure 4. Figure 4 represents the relationships between the action "Prepare a meeting" and related tasks. AND-OR graph is composed of four primitives:

- AND node, which represents that all of its inferior nodes should be selected. In Figure 4, AND node "Invitations" indicates that tasks "Make invitations" and "Send invitations" should be selected;
- OR node, which represents that only one of its inferior nodes should be selected appropriately. All branches under OR node are associated with the particular conditions. In Figure 4, OR node "Materials" indicates that the task "Print materials" can be selected when materials exist;
- The task, which is always placed at the terminal node. The tasks are shown as squares in Figure 4;
- The directed branch between nodes, which represents the execution order among tasks. In Figure 4, the task "Make invitations" should be executed before the task "Send invitations". When a directed branch is linked between non-terminal nodes, it represents that all inferior nodes under the source node must be executed before all inferior nodes under the destination node.

AND-OR graphs can associate with several actions. The inconvenience that an event generates only one action is avoided easily, using AND node. In Figure 4, the actions "Send invitations" and "Make materials" are collected together as AND node "Prepare a meeting".

3. Event Handler

We illustrate the structure of event handler in Figure 5. Our event handler is composed of four components:

- The event buffer holds temporarily events which have not been yet accepted. If some emergency events enter into this event buffer, the event handler accepts them preferentially;

- The knowledge is a database which accommodates the state transition and AND-OR graphs. The event handler extracts state transition and AND-OR graphs from the knowledge according to the event;
- The variable memory is a set of variables which indicate the current situation. The variables are used to select OR nodes in AND-OR graph, and updated by the event. If appropriate variables do not exist, the event handler asks to users;
- The working memory contains state transition graphs to represent the currently executing activities. With respect to events that may be received in the future, the event handler analyzes these events beforehand. For example, tasks related to the event "Receive responses of invitations" are generated before receiving the responses. The working memory contains AND-OR graphs and analyzed tasks.

Object Control

The task scheduler controls mutually related tasks dynamically. Of course, the task scheduler should manage the behaviors of objects(i.e. task executers).

1. Object and Task

Our objects are merely task executers which perform their methods concurrently, corresponding to the receiving messages, but do not give any effects to the behaviors of other objects. In the object-oriented paradigm [5,6,?], the interaction among objects is dependent on only one-to-one synchronous message passing. However, since in our model all interactions among objects are analyzed by the event handler from a static point-of view, our objects evaluate merely tasks as messages, being independent of other objects. To clarify the properties of objects, we define two types of methods:

- Reference method: When a reference method is executed, it returns the value corresponded to the object state. Since the reference method refers to the object state, the state after execution is the same as one before execution. This means that an object returns the same value when the same reference methods are executed successively.
- Update method: When an update method is executed, it returns the values "success" or "failure". If the update method is successful, the state before execution is altered to one after execution. If the update method fails, the state after execution is turned back to one before execution. This means that errors occurred in the update method are restored by the object itself.

With the above two-mode methods, every method in objects satisfies the following features:

1. The methods are defined as either reference or update methods;
2. All methods can not call the update methods of other objects;
3. The reference methods can not be called except that other objects are assigned as arguments.

These features make it possible that the task scheduler manages all properties of objects. For example, we consider the case that objects are allowed to update other objects. In this case, objects might change their states one after another when they execute their methods. If some methods fail, the task scheduler can not remove the effects of failed methods because it does not manage the successively changing object states. In the other word, all interactions among objects must be managed only by the event handler and task scheduler so that all objects should satisfy the above features.

The task is defined as a message that calls the update method and consists of the receiver, method name and some arguments. The task execution indicates to call the update method in the receiver, and at the same time to call all reference methods of arguments. Namely, the task execution changes only the state of receiver, but does not change the states of arguments.

2. Timestamp Ordering Method

We consider the case that it takes much time to execute particular tasks because of resource constraints(e.g. no allocation of memory). In this case, the task scheduler should execute the other executable tasks without waiting for the finishes of executing tasks. Namely, since the event handler determines the execution order of tasks only from a static point of view, this execution order is not always suitable to the dynamically changing execution situation. The task scheduler must reschedule interrelated tasks. Therefore, we introduce the timestamp ordering method, in which interrelated tasks are organized correctly by "timestamp"and the executable tasks are selected according to the assigned timestamps.

Figure 6 indicates four cases of the task execution related to the object X: $O(P)$ represents the execution of task whose receiver is the objet O(i.e. O is updated) and whose argument is the object P (i.e. P is referred), and $O(X) \rightarrow P(X)$ indicates that the task $O(X)$ is executed before the task $P(X)$. In Figure 6, the execution order of two tasks in the reference-reference relationship can be changed because these two tasks never conflict: if the execution is performed in the order of $P(X) \rightarrow O(X)$, the execution effect of O, P and X is the same as that of the execution in the order of $O(X) \rightarrow P(X)$. However, the task scheduler should never change the execution order among two tasks in the reference-update, update-reference and update-update relationships because two methods in these relationships always conflict: in the case of reference-update relationship, if the execution is performed in the order of $X(P) \rightarrow O(X)$, the execution effect of O and X is not the same as that of the execution in the order of $O(X) \rightarrow X(P)$. From such a consideration, the timestamps of newly generated tasks must satisfy the following properties:

Definition: Properties of timestamp

Assume that the task t_{pre} is a task which has already been assigned by its timestamp and not yet executed. The timestamp of newly generated task t_{new} must always satisfy the following properties:

1. If the receiver of t_{new} is either a receiver or argument of t_{pre}, then $TS(t_{new}) > TS(t_{pre})$;

2. If an argument of t_{new} is the receiver of t_{pre}, then $TS(t_{new}) > TS(t_{pre})$;
3. $TS(t_{new})$ and $TS(t_{pre})$ are always held as $TS(t_{new}) \geq TS(t_{pre})$.

where $TS(t_{new})$ (or $TS(t_{pre})$) indicates the timestamp of t_{new} (or t_{pre}) and ">"(or "$\geq$") represents the order between two timestamps. □

The properties 1. and 2. are derived from the relationships of execution order in Figure 9 and the property 3. is needed to assign the timestamps to all tasks. Using the timestamp satisfied with the above properties, our task scheduler always executes tasks which have the minimum timestamp.

3. Task Scheduler

We show the structure of task scheduler in Figure 7. Our task scheduler contains three components:

- A ready set, which contains tasks which have the minimum timestamp. These tasks are executed when all related tasks are not active. The task scheduler never sends messages to active objects. This indicates that the executions of some methods in the same object are controlled exclusively;
- A waiting set, which contains tasks which were taken out from the task queue. These tasks are associated with their own timestamps and ordered in the waiting queue according to their timestamps. If the ready set is empty, the task scheduler takes out the tasks with the minimum timestamp and creates a newly ready set;
- An object monitor, which always manages the activation of objects. The task scheduler controls objects exclusively, according to the information offerred by the object monitor.

Failure Handling

In general, the executing tasks do not always succeed: they sometimes fail. Because of unexpected failures or resource failures, the system may not accomplish the events completely. For example, if the task "Make invitations" fails, both the task "Send invitations" and the tasks related to the event "Receive responses of invitations" can not be executed. In the case that tasks fail, the alteration tasks must be prepared and the suspended tasks must be executed instantly. This function should be performed in the following steps:

- The failed tasks generate failure events((1) in Figure 8);
- The event handler determines both the undo and alteration tasks according to the failure events((2) in Figure 8);
- The task scheduler removes the effects of undo tasks and executes alteration tasks according to the requests of the event handler((3) in Figure 8).

1. Failure Event

The failure event indicates that the related task failed the execution, and generates the corresponding actions coped with the failure. In our prototype system, the failure event generates the actions which select the undo and alteration tasks by default. However, actions coped with failure are not always the same. Of course, it is useful for users to define different failure events in individual tasks. In order to realize such a definition, our task execution is performed as follows:

1. The task scheduler requests the execution to the task(by "execute" in Figure 9);
2. The requested task sends a message to its receiver object(by "perform:" in Figure 9);
3. The object received the message performs the corresponding update method and returns the value "success" or "failure"(by "true/false" in Figure 9).

In such an execution, the tasks can generate failure events without being dependent on other tasks or task scheduler. When a receiver object returns the value "failure", the related task generates the predefined event.

Smalltalk-80 can regard block statements as independent processes. Therefore, in our prototype system the task execution is controlled as described in Figure 10. In Figure 10, "[...] fork" represents that the block "[...]" is evaluated independently.

2. Failure Event Handling

The failure event is accepted preferentially by the event handler. The event handler accepted the failure event asks the failure handling way to users. Our handling way is classified into four classes:

1. Cancel only the failed task;
2. Alter the condition of OR node in AND-OR graph;
3. Alter the action which generates the failure task and then select a new AND-OR graph;
4. Cancel the state transition graph itself.

The undo tasks are determined when the user selects the above failure handling way. In addition, the alteration tasks are prepared when the above 2nd or 3rd way is selected. For example, we consider the case that the task "Send invitations as E-mails" fails because E-mail system crashed and that the user should change the condition of OR node. In Figure 11, when the condition of OR node "Send invitations" is changed(i.e. selection of ***demand = as_letters***), the tasks "Format invitations" and "Send invitations as E-mails" are determined as undo tasks, and the tasks "Print invitations" and "Send invitations as letters" are prepared as alteration tasks. These tasks are sent to the task scheduler with the failure handling request.

3. Rollback of Tasks

When the task scheduler receives the failure handling request, it must rollback undo tasks under the compensated procedure because all object states are successively changed whenever the related tasks are executed. Although various kinds of rollback means were proposed [4], we introduce a simple checkpoint mean as follows:

1. The task scheduler always preserves all consistent states of objects at appropriate intervals;
2. When the task scheduler receives the failure handling request, it turns all objects back to appropriate states according to the received request;

3. According to update and reference methods, the task scheduler removes the effects of undo tasks and then executes alteration tasks.

Figure 12 shows the algorithm for turning all objects back to appropriate states. In Figure 12, all objects are turned back to appropriate states and all tasks except undo tasks are selected as redo tasks. These selected redo tasks and alteration tasks are assigned with timestamps and ordered into the waiting queue. This way is not suitable but sufficient for the failure handling.

Conclusion

The object-oriented paradigm is not sufficient to control objects dynamically. In this paper, we regarded objects as task executers and established a concurrent control mechanism among interrelated tasks. In particular, we described the event handler which organizes the interaction among tasks from a static point of view, and the task scheduler which controls tasks on the basis of dynamic behaviors of them. These two components make it possible that our control mechanism can cope with the changing outside situations and resource constraints. Such an approach is sufficient for the flexible management of successive and interdependent tasks. In addition, we addressed the cooperation between the event handler and task scheduler in the case that the tasks fail. Although our proposed system is a first step to construct a secretary model, the state transition graph and AND-OR graph used by the event handler can deal with complex jobs, and also the timestamp ordering method performed by the task scheduler and failure handling way are valid to preserve the integrity constraint of objects.

Acknowledgements

We are very greatful to Prof.T.Fukumura of Chukyo University, and Prof.Y.Inagaki and Prof.J.Toriwaki of Nagoya University for their perspective remarks, and also wish to thank Ms.K.Sugino and our research members for their many discussions and cooperations.

References

[1] T.Ishida, L.Gasser and M.Yokoo:"Organization Self-Design of Distributed Production Systems", *IEEE Trans.on Knowledge and Data Engineering*, Vol.4, No.2, pp.123-133(1992).

[2] Y. Nakauchi, E. Miyoshi, T. Okada and Y .Anzai: "Computer-supported Cooperative Work Environments Based on Multi-agent Model", *Trans.on IEICE of Japan*, Vol.J75-D-II, No.11, pp.1874-1883 (1992) [in Japanese].

[3] K.-C.Lee, W.H.Mansfield Jr. and A.P.Sheth:"A Frameowrk for Controlling Cooperative Agents", *IEEE Computer*, Vol.26, No.7, pp.8-16 (1993).

[4] A.K.Elmagarmid(Ed.):"Database Transaction Model for Advanced Applications", *Morgan Kaufmann*(1992).

[5] J.Rambaugh:"Object-oriented Modeling and Design", *Prentice Hall*(1991).

[6] A.Goldberg and D.Robson:"Smalltalk-80: The Language and Its Implementation", *Addison-Wesley*(1983).

[7] B.Meyer:"Object-oriented Software Construction", *Prentice Hall*(1988).

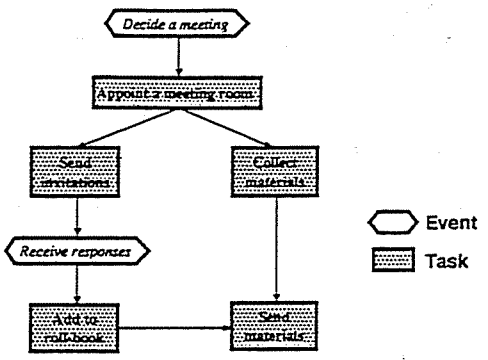

Figure 1: Tasks for meeting preparation

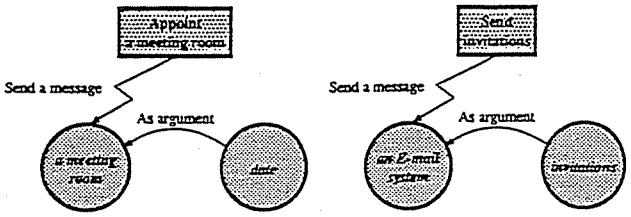

Figure 2: Correspondence between tasks and objects

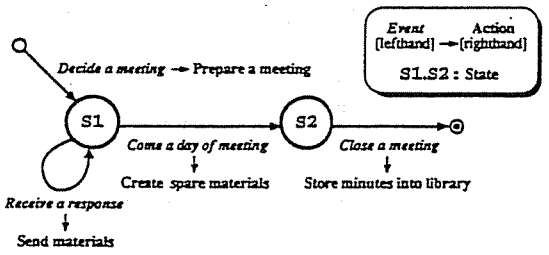

Figure 3: An example of state transition graph

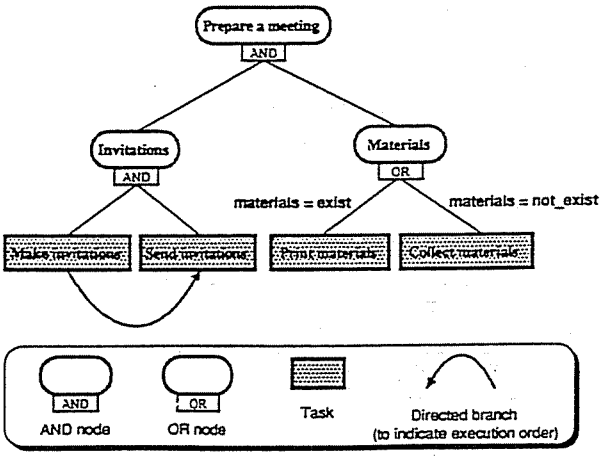

Figure 4: An example of AND-OR graph

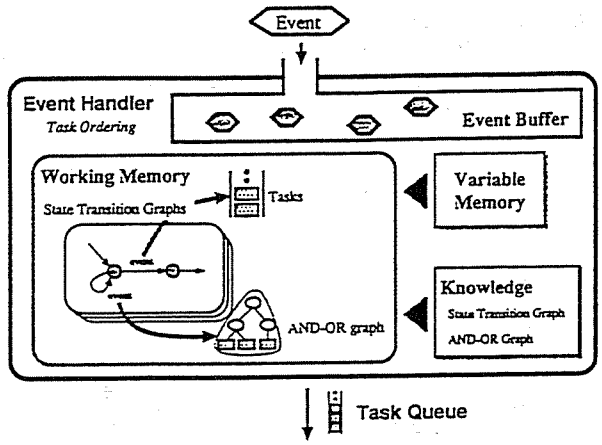

Figure 5: Structure of event handler

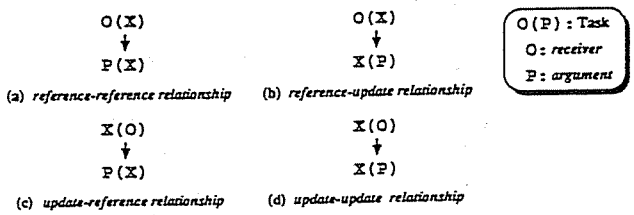

Figure 6: Execution order of two tasks related to object "X"

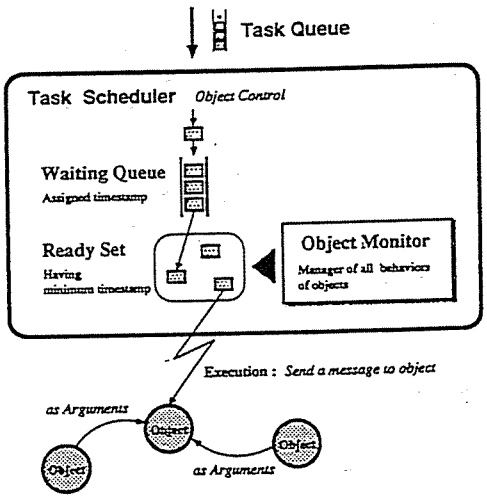

Figure 7: Structure of task scheduler

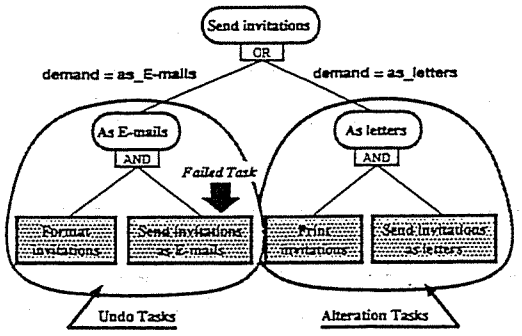

Figure 11: An example of task failure in AND-OR graph

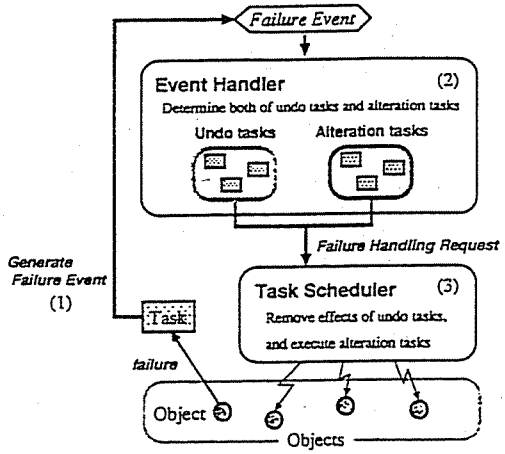

Figure 8: Failure handling

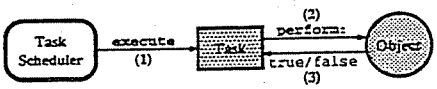

Figure 9: Task execution

```
Object subclass: #Task
    instanceVariables: 'receiver method arguments failureEvent...'
    classVariables: ''
    poolDictionaries: ''
    category: 'Secretary-System'!

!Task MethodsFor: 'execution'!

execute
    "execute the task"

    | value |
    [
    value := receiver perform: method withArguments: arguments.
    value = #failure
        ifTrue:
            [failureEvent generate]
  ] fork! !
```

Figure 10: Task execution on Smalltalk-80

```
procedure Turning all objects back to appropriate states.
  input    T_undo : a set of undone tasks.
           T_log  : an execution history including checkpoints.
  output   T_redo : a set of tasks that must be redone.
begin
  foreach t in Log (in order of recently executed tasks) do
    if t is included in T_undo then
      T_undo ← T_undo - {t}.
    else if t is checkpoint and T_undo is empty then
      Turning all objects back to checkpoint t.
      return
    else
      T_redo ← T_redo + {t}.
    endif
  endforeach
end
```

Figure 12: Algorithm for turning all objects back to appropriate states