Handling Uncertainties in Workflow Applications

Jian Tang
Department of Computer Science
Memorial University of Newfoundland
Canada
jian@cs.mun.ca

San-Yih Hwang
Department of Information Management
National Sun Yat-Sen University
Taiwan
syhwang@mis.nsysu.edu.tw

Abstract

Workflow techniques are currently seen as the key techniques to improve the effectiveness and productivity of business processes. Most workflow models take a static view toward workflow applications. The static view requires that all the components are known in advance and the structural information alone can uniquely determine an execution path. Some applications may not have either or both of these properties. In these applications, uncertainty exists. In this paper, we study the issues related to uncertainties. We characterize uncertainties into three categories, domain uncertainty, structural uncertainty and implementation uncertainty, and discuss the impact they have on workflow specification and implementation. We then propose approaches to coping with each of them.

1 Introduction

Workflow techniques are currently seen as the key techniques to improve the efficiency and productivity of business processes. Its importance in computer supported cooperative work has also been widely recognized by researchers in computer science/engineering community [11,12,13].

A *workflow* is a process consisting of a finite collection of well defined tasks. Usually the structure of the process is complex and the process itself is long running in nature. In addition, since business processes may have enormous differences in the way the tasks are organized, they may exhibit very different structures in terms of how the tasks are coordinated, executed and interacted. As a result, to guarantee the correct implementation of the semantics of workflow applications, it is vital to have a model that is powerful enough to support the specifications of not only the common features possessed by most workflow applications but also some special features arising from individual specifications. Recently, a number of models have been proposed [4,5,6,7,8,9,10,15,16] which may fit various applications.

One crucial requirement in a workflow specification is the identification of the requirements for the control/data flow among the participating tasks. In some literatures, these flows are supported by the concept of *dependency*. A dependency imposes certain restrictions on the interaction and/or execution of tasks. Traditionally, dependencies are implicitly specificed by the direct encoding into the program. However, in a complicated business application which may contain thousands of tasks, the portability and maintainability considerations render the direct encoding approach unattractive. This necessitates separate specification of these dependencies [1,3,17].

The early workflow models assumed that all the structural components could be specified in advance. While such a static view still applies to some (usually small) workflows which are static in nature, it is no longer appropriate for the others. In these applications, either the tasks may not be known in advance, or the dependencies may have to be selected and/or resolved at run time. We term this phenomenon *uncertainty*. (Another name used in the literature is *dynamic workflow*). As a consequence, any specification tool with sufficient generality must provide facilities to cope with uncertainties.

Some issues related to uncertainties have been studies by several researchers. In [15], the authors discuss methods to cope with the requirement for dynamically selecting dependencies. In [2], the authors propose a practical mechanism to deal with the applications where the dependencies between tasks may evolve dynamically. These studies, however, did not (or not intend to) characterize the problems. The coverage of the types of uncertainties is also limited.

In this paper, we study the issues related to uncertainties. We characterize the uncertainties commonly found in workflow applications into three categories: domain uncertainty, structural uncertainty and implementation uncertainty, and discuss the impact they have on workflow specification and implementation. We also suggest approaches to coping with them.

The rest of this paper is organized as follows. In Section 2, we introduce the basic concepts and give a high level view of the architecture. In Section 3, we introduce the three types of uncertainties and suggest new components to be incorporated into the model to represent them. In Section 4, we discuss some issues related to system support. We conclude the paper by summarizing the main results.

2 Workflows and the Related Uncertainties

2.1 Tasks and their dependencies

The smallest building blocks in the workflow model are tasks. A task specifies some work to be done. Tasks are carried out at local systems, and they cooperatively achieve a common goal. Exactly how a task is executed at a local system is transparent at the application level. The local systems are usually located at different sites. Thus the executions of the tasks are distributed. As a result, the coordination of the executions of different tasks is an important issue.

We distinguish a task from the program whose execution generates the task. For easy references, we will call a generating program the *task plan* for the corresponding task. A task plan must be *invoked* to generate a task. An invoker may be local or remote. A remote invoker initiates a task plan by issuing messages.

During its lifetime, a task may go through various externally observable states. Typical states include *not-executing*, indicating the task has not been started yet; *executing*, indicating that some of the operations of the task are still on going; *prepare*[1], signifying that all the operations of the task have been finished successfully; *commit*, signifying that all the operations in a task have been executed successfully and their effects have been irrevocably stored in the system; and *abort*, signifying the failure of the task, which requires that all the effects of the task be (semantically) eliminated.

A task may be invoked either by an external invoker or implicitly by the workflow management system. In either case, the constraints on the execution of various tasks, as specified by the user, must be enforced properly. These constraints can be realized by *state dependencies*. A state dependency specifies constraints on tasks' entering certain states. The following describes some common types of state dependencies.

Let s_1 and s_2 be two tasks:

1. if s_1 and s_2 both commit, then s_2 commits after s_1 commits–s_1 CFC s_2 (Commit-Follow-Commit);
2. s_2 can start only after s_1 commits–s_1 SBFC s_2 (Strong-Begin-Follow-Commit);
3. s_2 can start only after s_1 aborts–s_1 SBFA s_2 (Strong-Begin-Follow-Abort);
4. s_2 can commit only after s_1 aborts–s_1 SCFA s_2 (Strong-Commit-Follow-Abort);
5. If s_1 commits, then s_2 aborts–s_1 AOC s_2 (Abort-On-Commit);
6. If s_1 starts, then if s_2 has not committed yet, it must abort–s_1 WAOB s_2 (Weak-Abort-On-Begin);

The first dependency belongs to the type, *order* dependency. The types of the other dependencies are: for dependency 2, 3 and 4 *strong order* dependency, for dependency 5 *existential* dependency and for dependency 6 *weak existential* dependency. These types also apply to other combinations of states.

Note that we do not intend to exhaust all possible state dependencies in this paper. The above state dependencies are described simply because we will use them in the following examples. Besides, it is possible to come up with a high level language construct for specifying the dependencies of related tasks (e.g. Process Definition Language proposed by the Workflow Management Coalition [20]). In this case, the state dependencies can be used for internal processing.

From the way we composite these dependencies, we should read them as s_2 depends on s_1. For example, dependency s_1 WAOB s_2 should read 's_2 Weak-Abort-on-Begin depends on s_1'.

Example 1. Shown in Figure 1 are the tasks and their state dependencies in a typical workflow application: *travel planning*. Note the two dependencies between

[1] This state is not always provided. We include it here for generality.

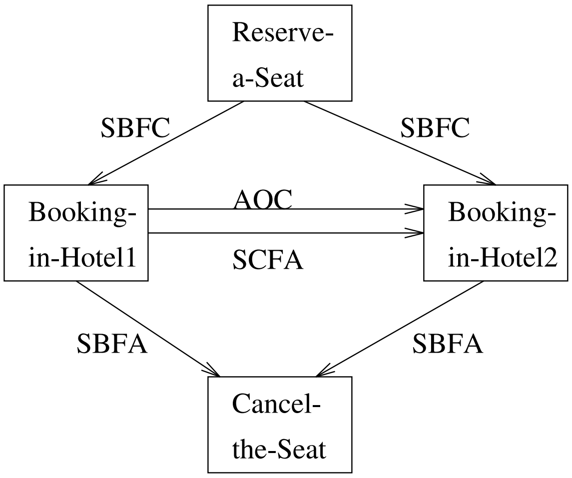

Figure 1: State dependencies in the travel planning example.

Booking-in-Hotel1 and *Booking-in-Hotel2*. They imply that at most one task can be successful. In addition, dependency SCFA implies that a room can be booked in Hotel2 only after no room is available in Hotel1. This dependency represents a preference given to Hotel 1. (This makes sense if, for example, the application is related to attending a conference which will be held in Hotel 1.)

2.2 Attributes of tasks

Using the states to model a task is adequate for the workflow applications where dependencies exist only between tasks of different types. However, we observe that in some workflow applications dependencies exist also between different tasks with the same type. (Refer to Example 2 in the next section.) These tasks, although generated by the same task plan, have different *attribute values*. The attributes of a task may include, for example, the invokers, the time when the task is generated, the final status (commit/abort) of the task, the location where the task will be executed, or even some output values. (For example, a task may output a special variable whose values indicate the degree of fulfillment of the expected goal.) For those workflow applications, some selected attributes of tasks must be explicitly modeled.

2.3 An architecture

Shown in Figure 2 is a hierarchical structure of a workflow system. On top of the hierarchy is the application. An application is usually written in some workflow language.

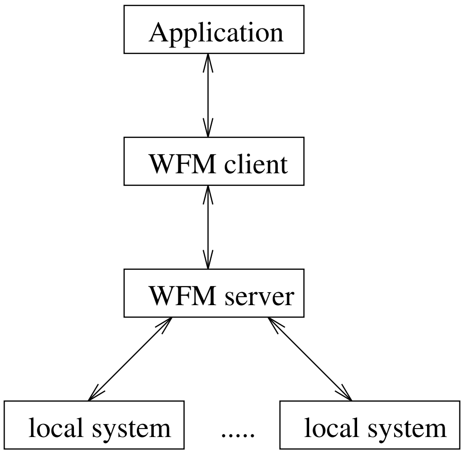

Figure 2: A high level view of system architecture.

Central to a workflow system is a *workflow manager* (WFM). The main function of the WFM is to coordinate the executions of tasks in such a way that the semantics of the workflow application is implemented. The WFM is usually organized as a client - server structure. The client initiates a workflow and as a result makes requests to the server. The server dispatches the tasks of the workflow to the appropriate local systems and wait for the responses from them. A response usually signifies a state transition of the task. The server itself can also initiate certain state transitions. (For example, from prepare to commit/abort, or from executing to abort, etc.) One of the vital functions performed by the server is to guarantee that no state transition violates the dependencies specified in the application. As mentioned before, local systems are responsible for carrying out the execution of the tasks. Local systems may be *autonomous* in terms of controlling state transitions of tasks. A fully autonomous local system will fully control the state transitions of tasks. On the other hand, some local systems may agree to wait for the commands from the WFM server before they make transitions.

3 Resolving Uncertainties in Workflow Applications

In the previous travel planning example, all the components are known in advance. For example, there are exactly four tasks, generated by one task plan each. The dependencies among these tasks are also completely specified. As mentioned before, there exist workflow applications where this is not the case. In this section, we look into this kind of applications and investigate their uncertainty property.

Based on the subcontext they appear in an application, we classify the uncertainties into three types, *domain uncertainty*, *structural uncertainty* and *implementation uncertainty*. In the following, we will use various examples to illustrate these concepts.

3.1 Domain uncertainty

This type of uncertainty happens in case where we cannot determine in advance how many tasks will participate in a workflow. The implication is that it is not always possible to name each task explicitly, i.e., by using conceptual symbols, like what we did in the travel planning example. This in turn implies that it is not always possible to specify dependencies among tasks in advance. The following example has been drawn from part of a live application, called PortNet[2], whose goal is to handle the visit by a foreign ship to Helsinki port.

Example 2: Relating to the visit of a foreign ship are various kinds of activities to be carried out by a number of department/organizations, such as the port, the customs office, the national navigation board, etc. The following are the activities at the port. A local agent first receives a notification, called *advanced notification*, from a foreign agent which contains various information about the ship's visit, such as the arrival date, the name of the ship, its radio call sign, length, width, etc., and the goods the ship is carrying. Since some information such as the arrival date is only an estimation in nature, the foreign agent will send the notifications repeatedly to update the information. Each time he/she receives a notification, the local agent initiates a task to process it. The application requires that if two tasks initiated by the agent both commit, then the one initiated later must commit after the one initiated earlier. This requirement ensures that the system state always reflect the up-to-date information. (For example, more accurate information always overwrites the less accurate information in the databases, but not the other way around.)

In addition to the agent, there is an appointed person at the port who *may* receive more accurate information from other (supposedly more reliable) sources, such as the captain of the ship. When this happens, the appointed person initiates an advanced notification task which processes the information he/she receives. This task will override those initiated by the agent.

When the ship has actually arrived, the local agent sends a *final notification* to the port which then initiates a task to process it. Once this task starts, any active tasks which process *advanced notifications* must stop.

Based on the specification, we have two task plans, one to generate tasks to process an advanced notification and the other to process the final notification. In addition, we must define three types of dependencies among the tasks: 1). For all the tasks initiated by the agent, those that process more recent notifications Commit-Follow-Commit-depend on those which process less recent notifications; 2). Among the tasks that handle advanced notices, the one initiated by the appointed person Commit-Follow-Commit-depends on those initiated by the agent; 3). Every task that processes the advanced notification Weak-Abort-On-Begin-depends on the task that processes the final notification (i.e., proc-final-noti WAOB proc-adv-noti). Note that the first dependency is defined on the tasks with the same type. So is the second dependency.

Since it is unpredictable how many times the local agent will receive the notifications from the foreign agent, we can not determine in advance how many tasks will be generated as a result of the local agent receiving the notifications. Consequently, we cannot specify the first two types of dependencies in a traditional manner, namely, in terms of the names of the tasks.

Note that a domain uncertainty would be a straightforward problem if there is not a need to identify the corresponding tasks *individually*. In Example 2, should we not have the first two types of dependencies, probably we would not care if those tasks can be determined in advance. Since the third dependency is defined only between the tasks of different task plans, its specification is not an issue: the task plans can be used to refer to the tasks unambiguously.

We introduce two additional components, namely *groups* and *sequences*, to resolve the domain uncertainty. A group is a set of tasks which share the same values for some attributes, such as invoker. A sequence is a set of ordered tasks based on the values of some attributes, such as the invocation time. With the introduction of these two concepts, it is possible to specify dependencies between the tasks generated by the same

[2] The first author participated in PortNet project during his visit to Technique Research Center of Finland (VTT) in 1994.

task plan. For example, to specify dependency 2 described above, we group all tasks invoked by the agent into group G_1 and all tasks invoked by the appointed person into group G_2. Then the dependency is simply specified as G_1 CFC G_2. To specify dependency 1, we arrange the tasks invoked by the agent into a sequence S based on the time they are invoked. We then specify the dependency as: for all i, $S(i-1)$ CFC $S(i)$.

One may argue that dependency 1 can be specified by a popular language construct provided by some workflow management products, namely workflow iteration (or called workflow loop). The workflow interation repeats the execution of a task plan until some condition is satisfied. However, the iteration primitive will actively invoke tasks while in Example 2, the tasks for processing advanced notifications are initiated passively by the local agent or the appointed person. Thus, workflow iteration is not suitable for resolving domain uncertainty.

Note that the techniques discussed above present only a general method to cope with domain uncertainty. When this model is represented specifically in an application, there may be more than one way to represent a group or a sequence.

3.2 Structural uncertainty

A structural uncertainty occurs when the types of dependencies that must be used in a workflow application can not be determined in advance. This type of uncertainty was described in [16]. In essence a structural uncertainty refers to a situation where some dependency may or may not actually be used at the runtime, depending upon certain conditions. Structural uncertainty is similar to the conditional routing described in the glossary of the Workflow Management Coalition (WfMC). However, since the state dependencies is more powerful in describing control dependencies than the control specification used by WfMC, the meaning of structural uncertainty is broader. The following is a simple example.

Example 3: Consider the application in Example 1. It is reasonable to expect that the attendants wish to have more flexibility in choosing hotels than the way described in the example. For instance, if the conference provides shuttle bus between Hotel2 and the conference site, then both hotels are equally acceptable to them. This means that we have two possible candidates for the set of dependencies between *Booking-in-Hotel1* and *Booking-in-Hotel2*:

1. *Booking-in-Hotel1* AOC *Booking-in-Hotel2*, *Booking-in-Hotel2* AOC *Booking-in-Hotel1*;

2. *Booking-in-Hotel1* AOC *Booking-in-Hotel2*, *Booking-in-Hotel1* SCFA *Booking-in-Hotel2*;

The first set should be used if there is a shuttle bus and the second be used if there is not. Exactly which one will be used is not known in advance. Thus the workflow specification must contain a *condition* to allow the selections.

A condition can be either internal or external. An internal condition is a predicate that contains states and/or input or output values of tasks. An external condition is based on the factors that are not directly related to any task. It is also possible that a condition is both internal and external, indicating that some variables in the condition are directly related to tasks but the others are not. The conditional components may take the general form:

condition 1: $D_{11}, \cdots, D_{1n_1}$;

$\cdots$

condition i: $D_{i1}, \cdots, D_{in_i}$;

where each condition is a combination of boolean expressions connected by such operators as and, or, etc., and each D_{ij} is a dependency. This structure means that if any condition is true, then all the dependencies under that condition must be enforced.

Example 4: The uncertainty described in Example 3 can be resolved by introducing the following conditional components:

¬*Hotel2-shuttle*: *Booking-in-Hotel1* SCFA *Booking-in-Hotel2*;

where *Hotel2-shuttle* is an external (boolean) variable which is true if and only if there is a shuttle bus between Hotel 2 and the conference site. The values of this variable can be filled out only at runtime.

Note although in general the conditions in a conditional structure can only be evaluated at runtime, we require that each condition not change throughout the execution, since otherwise incorrect execution may occur. Consider the following conditional component:

C_1: D_1;

C_2: D_2;

Suppose C_1 and C_2 compliment each other, and $D_1 \neq D_2$. During the time period when C_1 is true, D_2 may not be enforced. When C_2 becomes true later, it may turn out that D_2 has already been violated. This is inconsistent with the workflow specification.

The conditional components method described above requires that all possible conditions of interest be enumerated explicitly when the workflow application is specified. If the number of conditions is small, such a direct coding can improve the readability of the application program as well as the performance. However, if this number is large, as may be the case in large applications, enumerating all the conditions may render in a long and tedious specification, and therefore adversely affects the readability. In this case, we can use the dynamic workflow method proposed in [15]. Instead of enumerating all the conditions explicitly, an additional task is defined in the specification which will generate the appropriate structures at runtime.

Although we use the same term 'uncertainty', structural uncertainties differ from domain uncertainties in nature. A structural uncertainty reflects a fact that the workflow may choose to implement different application semantics at runtime. A domain uncertainty, on the other hand, is concerned only with the unpredictable nature of the instantiation of the activities in certain workflow applications.

3.3 Implementation uncertainty

This type of uncertainty refers to a phenomenon that there exists more than one choice of enforcing a dependency. Among the multiple choices, usually there is the most favorable one, the second most favorable one, and so forth. Note that in theory, implementation uncertainty exists for every dependency, since there always exists more than one enforcement method for any dependency. However, for some dependency, the best choice can be directly incorporated into the control structure when the workflow is specified. For instance, in Example 1, the following three resolution schemes are all correct for dependency *Reserve-a-seat* SBFC *Booking-in-Hotel1*:

1. After reserving a seat successfully, send a request to Hotel1 to book a room;

2. After reserving a seat successfully, never send a request to Hotel1;

3. Do not reserve a seat and do not request for booking a room;

The best choice is, of course, scheme 1. We can easily make this choice in advance by specifying a rule which requires that after Reserve-a-seat commits, Booking-in-Hotel1 be triggered. Thus the other two uninteresting cases are precluded. Now consider the dependency *Booking-in-Hotel1* SCFA *Booking-in-Hotel2*, which may run in parallel, still in Example 1. The following resolution schemes are all correct.

1. A negative response is received from Hotel1, after that a positive response is received from Hotel2 and accepted.

2. A positive response is received from Hotel1 and accepted, after that a positive response is received from Hotel2 but rejected;

3. A positive response is received from Hotel1 and rejected, after that a positive response is received from Hotel2 and accepted;

Evidently, from the user's intention in using dependency SCFA, he/she is in favor of Hotel1. Thus scheme 2 is the best choice.

To make the matter more complicated, the following example shows that the best choice may vary depending upon some conditions.

Example 5: Considering Hotel1 to be always more favorable than Hotel2 may risk delaying the entire trip if the negative response from Hotel1 comes very late. A safer scheme would be to set a deadline (for example, one week before the departure) for the response from Hotel1. If no response has been received from Hotel1 by the deadline, then thereafter we will accept whichever offer comes first.

To meet the above requirement, we may first specify the dependencies:

Booking-in-Hotel1 AOC *Booking-in-Hotel2* and *Booking-in-Hotel2* AOC *Booking-in-Hotel1*.

Then among the three possible choices:

1. Book a room in Hotel1 but not in Hotel2;

2. Book a room in Hotel2 but not in Hotel1;

3. Do not book rooms in either hotels.

we choose the first to be the best before the deadline and the first two to be equally acceptable on or after the deadline. The third choice is made only if neither the first nor the second is possible. Clearly, it would be difficult, if not impossible, for the runtime support to make the above choice without being told in advance the user's intention. To cope with this problem, we propose to incorporate 'hints' from the end-users into the workflow specification. Each hint is associated with one state dependency. Generally speaking, a hint specifies a guideline for enforcing the associated dependency under various conditions. The following is the general form of a hint:

condition 1: $T_{11}, \cdots, T_{1n_1}$;
$\dots$
condition i: $T_{i1}, \cdots, T_{in_i}$;

The conditions have the same meaning as those in the conditional components introduced in the last section. T_{ij} is a set of pairs of task and states. Following each condition is a priority list in descending order. When a decision is to be made, we always try to choose one from the set with the higher priority. We move to the set with the next lower priority only if the one with the higher priority will invalidate the associated dependency.

Example 6: In Example 5, to resolve the implementation uncertainty stated there, we give hints associated with dependency *Booking-in-Hotel1* AOC *Booking-in-Hotel2*:

- t <*DepartureTime* - one_week: <{*Booking-in-Hotel1.commit*},{*Booking-in-Hotel2.commit*}>;
- *DepartureTime* - one_week≤t: <{*Booking-in-Hotel1.commit*, *Booking-in-Hotel2.commit*}>;

The first row indicates that if the positive response from Hotel2 comes more than one week before the departure date, we should not accept it, since otherwise we would have given up trying the first choice. The second row states that if the time is later than one week before the departure date, then we can accept a room from whichever hotel offers it first.

It may seem that we can use the conditional component approach in Section 3.2 to solve the implementation uncertainty problem. For example, under the first condition, instead of giving the preferred states explicitly, we may use dependency *Booking-in-Hotel1* SCFA *Booking-in-Hotel2*. However, this does not prevent the system from rejecting a positive response from Hotel1. In general, conditional component approach is inappropriate for implementation uncertainty since it is essentially a specification mechanism which is difficult, if not impossible, to control the way the specification is implemented. Another reason that the conditional component approach is not suitable for implementation uncertainty is seen in the next paragraph.

Note that a hint serves only as a message delivered from the workflow users to instruct the WFM how to choose a preferred resolution scheme. If there is ever a conflict between a hint and the associated dependency, the latter must override the former. An implication of this is that if a hint contains different conditions, we allow these conditions to change truth values over time. (This will increase the flexibility in developing workflow applications.) If under the new truth value, the hint conflicts with the dependency, then the system will ignore the hint and respect the dependency. For instance, suppose a room is booked in Hotel2 seven days before the departure date. Also suppose that for some emergent reasons, the departure has to be delayed until the next day. (Thus new reservation must be made.) If at this time the positive response comes from Hotel1, it must be rejected to respect the AOC dependency is Example 6.

4 Discussion

Although this paper is concerned only with application level, the way various concepts is supported by WFM is an important issue. The most important goal of the WFM is to ensure that the dependencies between tasks are correctly enforced at runtime. This requires the cooperative efforts of different parts of the WFM. When an application is initiated, the WFM client sends a request to the WFM server. The server will then create a task by interacting with the appropriate local system. If a state transition of a task is needed, the server determines which transition to make by consulting the hint and by ensuring that the supposed transition will not invalidate the dependencies related to the task.

We have proposed a workflow language called C^{++}WF, which combines the object-oriented features from C++ with a rule based macro language. Users can use C^{++}WF to describe the state dependencies of tasks in a workflow. It also supports the features for resolving uncertainties described in this paper. When a workflow is instantiated, a thread that executes the corresponding C^{++}WF program is generated. This thread controls the execution of the related tasks in local systems and reports the execution status to the WFM client upon request. Due to space limitation, we do not describe C^{++}WF in detail in this paper. Interested readers are referred to [18] for the description of the language. C^{++}WF is currently under development at MUN.

Another issue is how to enforce a dependency. The enforcement of a dependency requires that its semantics never be invalidated at any point throughout the life time of the workflow. As seen before, some dependency can be directly implemented in the application in terms of rules. For the others, separate enforcement mechanisms must be employed. In general, dependency enforcement is a complicated issue. Although some works have been reported [1,21], more efficient solutions are still neeeded.

5 Conclusion

In this paper, we discuss uncertainty problems that may be present in workflow applications. Based on their nature and the subcontext they appear, we classify these uncertainties into three types, domain uncertainty, structural uncertainty and implementation uncertainty. We then suggest approaches to resolving them. The main idea underlying these approaches is to incorporate more semantical components into a workflow model such that uncertainty can be represented. We also discuss some issues about the system support.

We are currently developing and implementing a workflow language C^{++}WF, which incorporates features for handling uncertainties described in this paper. The experience and lessons we learned will be reported in the future.

REFERENCES

1. M. Attie, M. Singh, A. Sheth, and M. Rusinkiewicz, "Specifying and enforcing intertask dependencies", Proc. of the 19th Intl. Conf. on VLDB, 1993.
2. D. Barbara, S. Mehrota and M. Rusinkiewicz, "INCAS: a computation model for dynamic workflows in autonomous distributed environments", Tech. Report, Matsushita Information Technology Lab., April 1994.
3. Y. Breitbart, A. Deacon, H. Schek, A. Sheth and G. Weikum, "Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows", SIGMOD Record, 22 (3), September 1993.
4. O. Bukhres, A. Elmagarmid, and E. Kuhn, "Implementation of the flex transaction model", IEEE Data Engineering Bulletin, 16(2), June 1993.
5. P. Chrysanthis and K. Ramamritham, "A formalism for Extended Transaction Model", Proc. of the 17th VLDB, 1991.
6. U. Dayal, M. Hsu, and R. Ladin, "A transaction model for long-running activities", Proc. of the 17th VLDB, 1991.
7. A. Elmagarmid (ed.), "Database transaction models for advanced Applications", Morgan-Kaufmann, 1992.
8. A. Elmagarmid, Y. Leu, W. Litwin and M. Rusinkiewicz, "A multidatabase transaction model for InterBase", Proc. of the 16th VLDB, 1990.
9. H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, "Modeling long-running activities as nested sagas", IEEE Data Engineering Bulletin, 14(1), Mar ch 1991.
10. D. Georgakopoulos and M. Hornich, "An extended transaction environment for workflows in distributed object computing", in [14].
11. D. Georgakopoulos, M. Hornich,"A framework for enforceable specification of extended transaction models and transaction workflows", International Journal of Intelligent and Cooperative Information Systems, World Scientific, Sept., 1994.
12. J. Grudin,"Computer supported cooperative work: history and focus", IEEE Computer, May 1994.
13. M. Hsu (ed.), "Special issue on workflow and extended transaction Systems", Bulletin of the Technical Committee on Data Engineering (IEEE Computer Society), 16(2), June 1993.
14. J. Klein, "Advanced rule driven transaction management", Proc. of the IEEE COMPCON'91.
15. N. Krishnakumar and A. Sheth, "Managing heterogeneous multi-system tasks to support enterprise-wide operations", Distributed and Parallel Databases, 3, 1995, pp 155 - 186.
16. F. Leymann and W. Altenhuber, "Managing Business Processes as Information Resources", IBM Systems Journal 33(2), 1994.
17. D. McCarthy and S. Sarin, "Workflow and transactions in InConcert", Bulletin of the Technical Committee on Data Engineering, IEEE Computer Soci
18. J. Tang and S.-Y. Hwang, "Handling Uncertainties in Workflow Applications", Tech. Report, Department of Computer Science, Memorial University of Newfoundland, Canada.
19. J. Tang and J. Veijalainen, "Enforcing inter-task dependencies in transactional workflows", Proc. of 3rd Intl. conf. on Coopis, 95, pp 72 - 86.
20. Workflow Management Coalition, "Workflow Management Coalition Interface 1: Process Definition Interchange," Document Number: WfMC TC-0020 (Draft 5.0), Feb. 3, 1996.