A Reflective Framework for Capturing Complex Dependencies of Workflow Activities

Ling Liu*
Dept. of Computing Science
University of Alberta
GSB 615, Edmonton, Alberta
T6G 2H1 Canada
lingliu@cs.ualberta.ca

Calton Pu†
Dept. of Computer Science
Oregon Graduate Institute
P.O.Box 91000 Portland
Oregon 97291-1000 USA
calton@cse.ogi.edu

Abstract *Complex workflow activities are typically of hierarchically nested structure and sophisticated interaction dependencies, and often are vulnerable to uncertain duration and unpredictable development. Methods and techniques that can avoid or reduce any significant performance loss due to such uncertainty and unpredictability are proved to be desirable and rewarding. In this paper we describe a reflective framework for declarative specification and dynamic restructuring of complex workflow activities, including the transactional activity composition model (TAM) and a family of activity-split and activity-join meta operations. By using the TAM activity specification facility, both the hierarchical organization of complex activities and the wide spectrum of interaction dependencies among concurrent activities can be captured declaratively and incrementally, allowing reasoning about correctness of complex activities independently from their transaction implementation mechanisms. By using the set of activity-split and activity-join meta operations introduced, users may restructure ongoing activities, in anticipation of uncertain duration or unpredictable development, to allow performance improvement through release of early committed resources or transferring ownership of uncommitted resources.*

1 Introduction

Since 1985 [2] significant results have been produced in the area of database support for collaborative applications. At the same time, the continual expansion of network bandwidth and increase in computing speed raised both the quantity and complexity of data types. In addition, the growing sophistication and intricacy of collaborative applications, such as CASE tool designs, CAD/CAM projects, and business workflow information systems, pushed the limits of previous results on transactional support for cooperative workflow activities. Of the three main characteristics [21] of the cooperative activities, the first two, *uncertain duration* and *unpredictable development*, are aggravated by the increased necessity of accessing and sharing data over widely distributed information sources, and the third characteristics, *flexible interaction* among concurrent activities, has become an increasingly higher challenge due to the growing complexity of interactions found in collaborative applications. Therefore, in many enterprise-wide workflow applications, it is not only difficult to foresee the most effective composition structure of complex activities at early stage of the design, but also most of the cooperative workflow tasks may not even have a single hierarchical structure that is effective at all time, because of the unpredicability in the state of network configuration and the unexpected unavailablity of data sources.

To address the challenges outlined above, we present the Transaction Activity composition Model (TAM) and a set of activity-split and activity-join operations as a unified framework for declarative specification and dynamic restructuring of activities of sophisticated interactions. TAM can be seen as a careful combination of the compositional activity model proposed in [14] with well defined extended transaction models such as split/join transactions [21]. Our results reported in this paper are significant advance over each one of the component technology we used and over the previous activity models. First, TAM offers a rich vocabulary and a modular design method for modeling and reasoning about the whole spectrum of interactions among activities and among cooperating ob-

*Supported partially by NSERC grant OGP-0172859 and NSERC grant STR-0181014
†Supported partially by NSF grant IRI-9510112.

jects in a multitask workflow environment. Second, TAM provides a set of meta operators for composing and restructing activities through various behavioral refinement and behavioral composition facilities (including various activity-split and activity-join operators). The use of the TAM refinement and composition operators not only allows performance improvements through early release of locked resources but also supports the reasoning about concurrency and recovery properties of cooperative activities. Third, TAM separates the specification of activity patterns from their transaction implementation strategies, thus allowing more reasoning and optimization to improve performance in the actual implementation.

Another distinct feature of the proposed framework is its unique application of computational reflection concept and principles [13] throughout the development of TAM and its associated set of activity restructuring operators. The notion of a reflective framework is employed to address the following two open problems in transactional support for complex activities: (1) providing specification interface for flexible customization and (2) providing both application-level and system-level of customization to allow changes in requirements and in actual behavior of the system to be adapted seamlessnessly.

TAM's potential application areas include: (1) workflows involving tasks that share multiple heterogeneous databases; (2) workflows that need a definable and provable "correct" execution of the workflow as determined by the intertask dependencies in the presence of concurrently executed tasks of different invocations of the workflow; (3) multi-user design environments where not only customers may invoke "competing" service requests simultaneously, but also service providers need to cooperate in accomplishing a service provision; (4) applications consisting of activities that are open-ended and cooperative in nature, and thus there is a demand for supporting well-defined activity restructuring operations such that a "correct" cooperation of dynamically split or joined tasks of the workflow is guaranteed.

2 The TAM Model

The basic building block of TAM is activities, which are described by *activity patterns*. An activity pattern describes the communication protocol of a group of cooperating objects in accomplishing a task [14]. Informally, an activity pattern consists of objects, messages among them, constraints, preconditions, postconditions, and triggering conditions of executions. TAM distinguishes two types of activity patterns: *elementary activity patterns* and *composite activity patterns*. An elementary activity pattern is a program that issues a stream of messages to access a underlying database. We refer to an execution of an elementary activity pattern an *elementary* activity. A composite activity pattern consists of a tree of activity patterns and a set of activity dependencies. The tree organization of an activity pattern α is called *activity hierarchy* of α. We call an execution of a composite activity pattern α a *composite activity* and an execution of a constituent activity pattern a *subactivity*.

In the sequel we use the term *activity* (or *subactivity*) and the term *activity pattern* (or *constituent activity pattern*) interchangeably when no conflict incurs.

2.1 Characterization of Dependencies

There are two generic types of dependencies among activities.

- *Architectural dependencies* (or *structural dependencies*), which reflect the hierarchical organization of the system into abstract data types of increasing complexity.
- *Correctness dependencies* (or *dynamic dependencies*), which arise from the use of shared data and typically specify what kind of execution schedules of activities preserves data consistency and produces correct results, thereby defining an application-dependent correctness criterion.

The architectural dependencies reflect the *invocation hierarchy* of the system, whereas the dynamic or correctness dependencies reflect the intended *behavioral interactions* (e.g., interleavings) among any number of activities that are unrelated in terms of the invocation hierarchy. Note that dynamic dependencies are not at all hierarchical [16]. Any activity (incl. atomic action) at any invocation level can be involved in dynamic (correctness) dependencies with many other activities at different levels of the invocation hierarchy. In TAM, the set of *architectural dependencies* and *correctness dependencies* are considered as the *baselines* that are required to meet *in the breakdown of a complex task* and *in the concurrent executions of a complex workflow activity*. More concretely, these activity dependency rules are viewed as integrity constraints on the executions of activities to ensure that the activities executed concurrently interact only in the ways *allowable* by its correctness criteria and thus leave the database in a consistent state.

Orthogonal to the two generic types of activity dependencies, for each activity T we can alternatively classify the activity dependencies associated with T into two kinds: intra-activity dependencies and inter-activity dependencies. *Intra-activity dependencies* define the behavioral relationships between (direct or indirect) subactivities of T. *Inter-activity dependencies* define the behavioral relationships between T and all activities that are not subactivities of T. Intra-activity dependencies can be either architectural or correctness (dynamic) dependencies, whereas all inter-activity dependencies are correctness (dynamic) dependencies.

In this paper we discuss activity patterns primarily in terms of the set of constituent activities and the set of intra-activity dependencies.

2.2 Correctness Dependencies

Activity correctness dependencies arise from the use of shared data and specify which concurrent executions (schedules) of activities preserve data consistency and produce correct results. The activity correctness dependencies currently being considered in TAM include:

- **state transition dependencies**, which define state transition conditions on the execution structure of activities.
- **execution precedence dependencies**, which specify the application-dependent semantics of intended execution orders of a set of activities (e.g., activity A_1 precedes activity B).
- **serialization dependencies**, which specify whether the concurrent executions of a set of activities on a set of objects must be equivalent to one of the intended (desired) activity execution orders that are known to preserve data consistency and produce correct results.
- **recovery dependencies**, which specify the triggering conditions that can take the database to a state that is considered correct (consistent) in the event of failure.
- **visibility dependencies**, which define the cooperation conditions in which an activity can see (access) the results of some other activities while it is executing (e.g., a child activity has access to all objects in $\mathcal{O}$ which its parent can access).

The temporal precedence dependencies and serialization dependencies are also referred to as *activity execution dependencies* because they present the dynamic constraints on the execution structure of activities.

Important to note is that, given an activity pattern (say α), the complete set of activity correctness dependencies associated with α include both the dependencies that are *explicitly* defined by workflow activity designers and the dependencies that are *implicitly* defined through the construction of the activity composition hierarchy of α.

The implicit activity dependencies described by an activity hierarchy include not only the architectural dependencies but also the correctness dependencies. The implicit architectural dependencies, once specified, can only be modified through dynamic activity-split and activity-join operators. The implicit correctness dependencies can be overriden by any activity dependencies that are explicitly defined by the transaction or workflow activity designer.

2.3 A Workflow Scenario

Consider a composite activity TELECONNECT that performs telephone service provision task by installing and billing of telephone connection for a telecommunication company. This example is an extended scenario of the workflow application for designing telephone circuit described in [1, 10]. Suppose the activity A:TELECONNECT consists of four subactivities A_1:CLIENTREGISTER, A_2:CREDITCHECK, B:ALLOCATECIRCUIT, and A_3:INSTALLNEWCIRCUIT (see Figure refAltDesign2(a)).

A is executed when a telephone company customer requests telephone service installation. Activity A_1 registers the client information in the client database for billing purpose and pass a service order to the activity B. Activity A_2 evaluates the credit history of the client. Activity B:ALLOCATECIRCUIT(ClientId, Start, End, Circuit-Id) attempts to provide a connection by allocating existing resources, such as allocating lines (C:ALLOCATELINES), allocating slots in switches (A_8, A_9), and preparing bill for establishing the connection (A_{10}). The activity of allocating lines (C) in turn has a number of subtasks such as selecting nearest central offices (A_4), relocating existing lines (A_5:ALLOCATELINE1, A_6:ALLOCATELINE2) and spans (trunk connection) between two allocated lines (A_7:ALLOCATESPAN). If B succeeds, the cost of connection is minimal. The activity A_3:INSTALLNEWCIRCUIT is designed to performs an alternative task that involves physical installation of new facilities in the event of failure of activity B.

Figure 1 and Figure 2 present the TAM activity specifications for this example. As you may see, each activity is described by specifying the list of children activities and their interaction dependencies through the behavioral aggregation facility, thus all the composite activities are defined in a declarative and incremental fashion.

This example also indicates that the activity composition hierarchy (see Figure 3(a)) presents some implicit dependencies between an activity and its children activities, whereas the dependencies explicitly specified in each activity pattern (see Figure 1 and Figure 2) are application-specific activity interaction dependencies. TAM classifies application-specific interaction dependencies into three categories: activity execution dependencies, activity interleaving dependencies, and activity state transition dependencies. In subsequent sections we will first discuss the implicit dependencies defined by activity composition hierarchy and then discuss application-specific dependencies and their roles in specification and verification of correctness of concurrent executions of complex activities.

2.4 Implicit Activity Dependencies

Implicit activity dependencies refer to the dependencies between an activity and its (direct or indirect) subactivities, and are defined implicitly through the activity composition hierarchy. We below discuss two important types of implicit activity dependencies: default activity state transition dependencies and activity visibility dependencies.

2.4.1 Activity State Transition Dependencies

Each TAM activity has a set of observable states $\mathcal{S}$ and a set of possible state transitions ψ: $\mathcal{S} \longrightarrow \mathcal{S}$, where $\mathcal{S} = \{$*begin, commit, abort, done, compensate*$\}$. The transitions between the states of an activity can be represented by a finite state automation and its transition graph. When an activity T is activated, it enters the state *begin* and becomes active. The state of T is changed from *begin* to *commit* if T commits, and to *abort* if T or its parent aborts. If T's parent aborts after T is committed, then its state is changed to *compensate*; if T's root activity commits, then its state becomes *done*. When T is a composite activity, T enters the *commit* state if all its component activities A_i $(i = 1, ..., n)$ legally terminate (i.e., commit or abort). In other words, the activity T may commit even if one of its component activities is aborted.

2.4.2 Activity Visibility Dependencies

TAM uses the multiple object version scheme [20] to define the notion of visibility in the presence of concurrent execution of activities, that is, object versions are created and maintained at different points in the activity hierarchy. Each activity has a local set of object versions, which is accessible to any of its children subactivities in the hierarchy. Thus, there may be several versions of an object scattered throughout the activity hierarchy. For a specific object, the version at the *Root* activity is the oldest version, and the ones further down in the hierarchy are causally dependent on it. With multiple object version scheme, a notion of visibility is defined naturally according to the hierarchy. For a given activity T, when an object in T's data access set receives an access request from a child subactivity of T, a new version is automatically created for this subactivity. The new version of the object is written back to the parent activity's access set when the subactivity that initiated the access to the object commits or comes to a breakpoint. A *breakpoint* [5, 9] of an activity is a point in its execution where it can interleave with other activities. With the multiple object version scheme, the *Root* activity at the top of the activity hierarchy contains the most stable version of each object, and guarantees that it can recover its copies of objects in the event of system failure.

2.5 Application-specific Dependencies

In the domain of cooperative applications, the notion of what is correct may vary from application to application, from task to task, and from user to user. There is no single correctness criterion like global atomicity for concurrent executions of activities, because the global correctness criteria such as atomicity were inapplicable in an enterprise networking environment that supports complex activities and encourages cooperation rather than competition with respect to resources. Hence we should allow user-defined and application-dependent correctness specification. This prompted us to look for mechanisms for specification and enforcement of user-supplied programmable correctness criteria.

Temporal precedence and cooperative constraints are two typical correctness specification for complex ac-

begin Activity TeleConnect(*In:* ClientId:CLIENT, Start:POINT, End:POINT, *Out:* CircuitId:CIRCUIT)

Behavioral Aggregation of component Activities:
A_1: ClientRegister(*In:* ClientId:CLIENT, Start:POINT, End:POINT)
A_2: CreditCheck(*In:* ClientId:CLIENT, Start:POINT, End:POINT, *Out:* creditStatus:Boolean)
A_3: InstallNewCircuit(*In:* ClientId:CLIENT, Start:POINT, End:POINT, *Out:* CircuitId:CIRCUIT)
B: AllocateCircuit(*In:* ClientId:CLIENT, Start:POINT, End:POINT, *Out:* CircuitId:CIRCUIT)

Execution Dependencies:
$ExeR_1$: A_1 **precede** B, A_3

Interleaving Dependencies:
ILR_1: A_2 **precede** A_{10}

State Transition Dependencies:
STR_1: *abort*(B) **precede** *commit*(A_3)
STR_2: *commit*(B) **disable** A_3
STR_3: *commit*(B) **disable** *begin*(A_3)

end Activity

Figure 1: An example specification of composite activity TeleConnect

tivities. In contrast to traditional serializability theory [4], temporal precedence correctness criteria consider histories *correct* if and only if they are equivalent to some specific serial history [11]. Cooperative correctness criteria [5, 9, 17, 19, 10] use less restrictive notions of conflicts that take into account application-dependent semantics, and allow *compatible* activities to cooperate, e.g., to repetitively read and write specific objects without restrictions.

In TAM two activities are defined to be *compatible* if and only if their execution order does not matter in terms of the application-specific activity dependencies specifications. When the execution order of two subactivities within a composite activity T is important to the correctness of concurrent executions of T, we call these two subactivities *incompatible* w.r.t. T. As subactivities executing concurrently may interact with each other in undesirable ways, we need a way for the composite activity to enforce that *incompatible* subactivities will be isolated from each other in the concurrent executions to prevent unwanted side effects. In TAM a compatibility specification of activities is captured in the form of a compatibility matrix [16].

TAM provides four constructs for specification of user-defined activity dependencies. They are **precede, enable, disable**, and **compatible**. The construct **precede** is designed to capture the temporary precedence dependencies and the existence dependencies between two activities. For example, "A **precede** B" specifies a *begin-on-commit* execution dependency between the two activities: "B cannot begin before A commits". The constructs **enable** and **disable** are utilized to specify the enabling and disabling dependencies between activities. One of the critical differences between the construct **enable** or **disable** and the construct **precede** is that **enable** or **disable** specifies a triggering condition and an action being triggered, whereas **precede** only specifies an execution precedence dependency as a precondition that needs to be verified before an action can be activated, and it is not an enabling condition that, once satisfied, triggers the action. The construct **compatible** declares the compatibility of activities A_1 and A_2. It is provided solely for specification convenience since two TAM activities are compatible when there is no execution precedence dependency between them.

Recall the telecommunication example given in Section 2. The following is a set of application-specific activity dependencies between AllocateCircuit and InstallNewCircuit specified in Figure 1:

- (1)A_1 **precede** B, A_3.
  (B, A_3 cannot begin before A_1 commits)
- (2)*abort*(B) **precede** *commit*(A_3). (A_3 cannot commit before B aborts)
- (3) *commit*(B) **disable** A_3.
  (A_3 must abort if B commits)
- (4) *commit*(B) **disable** *begin*(A_3). (A_3 cannot begin after B has committed)

Note that (3) and (4) cannot be replaced by the following state transition dependency "A_3 cannot commit after B commits", i.e., "*commit*(B) **disable** *commit*(A_3)", because the later allows A_3 to continue its execution, holding resources or performing unnecessary computations after B has committed. Note also that the dependency (2) implies that A_3 can only commit after B *abort*s, whether A_3 eventually commits or not is an orthogonal matter.

begin Activity ALLOCATECIRCUIT(*In:* ClientId:CLIENT, Start:POINT, End:POINT, *Out:* CircuitId:CIRCUIT)

Behavioral Aggregation of component Activities:
C: ALLOCATELINES(*In:* Start:POINT, End:POINT, *Out:* CircuitId:CIRCUIT)
A_8: ALLOCATESWITCH(*In:* Line_1:LINE, *Out:* Span)
A_9: ALLOCATESWITCH(*In:* Line_1:LINE, *Out:* Span)
A_{10}: PREPAREBILL(*In:* ClientId:CLIENT, Line_1:LINE, Line_2:LINE, Span, *Out:* CircuitId:CIRCUIT)

Execution Dependencies:
$ExeR_1$: $\{C, A_8, A_9\}$ **precede** A_{10}

Interleaving Dependencies:
ILR_1: $\{A_5, A_7\}$ **precede** A_8
ILR_2: $\{A_6, A_7\}$ **precede** A_9

State Transition Dependencies:
STR_1: $abort(C) \vee (\forall A_i\ (i = 8, 9)\ abort(A_i)$ **enable** $abort(self)$

end Activity

begin Activity ALLOCATELINES(*In:* Start:POINT, End:POINT, *Out:* CircuitId:CIRCUIT)

Behavioral Aggregation of component Activities:
A_4: SELECTCENTRALOFFICES(*In:* Start:POINT, End:POINT, *Out:* Off_1:CentralOff, Off_2:CentralOff)
A_5: ALLOCATELINE(*In:* Start:POINT, Off_1:CentralOff, *Out:* Line_1:LINE)
A_6: ALLOCATELINE(*In:* End:POINT, Off_2:CentralOff, *Out:* Line_2:LINE)
A_7: ALLOCATESPAN(*In:* Off_1:CentralOff, Off_2:CentralOff, *Out:* Span)

Execution Dependencies:
$ExeR_1$: A_4 **precede** $\{A_5, A_6, A_7\}$

State Transition Dependencies:
STR_1: $\forall A_i\ (i \in \{4, \ldots, 7\})\ abort(A_i)$ **enable** $abort(self)$

end Activity

Figure 2: An example specification of activities ALLOCATECIRCUIT and ALLOCATELINES

3 Restructuring of Activities

In a cooperative environment, activities reveal two characteristics intrinsic to distributed (and possibly long-running) transactions. First, initial activities may sometimes modify data early in the transaction and will then remain unchanged until commit. Second, some or all of the later activities (subtasks) may become independent of previous activities (subtasks). In both cases, it is desirable for the results of earlier activities to be made available to other *concurrent* activities before end of the long-running transaction, to increase the concurrency and avoid introducing the possibility of cascaded aborts.

3.1 An Example

Recall the composite activity TELECONNECT discussed in Section 2.

The designer of the activity ALLOCATECIRCUIT may later decide to unnest the ALLOCATELINES activity (dotted balloon in Figure 3(a)), through "*split-by-unnest*" operation, with the result shown in Figure 3(b), in order to add concurrency and improve cooperation among subactivities $A_5, A_6, \ldots, A_9$. In the initial design of Figure 3(a), activity A_6, which allocates a line between the central office Off_2 and the *end* point of the requested connection, has nothing related to the allocation of $Switch_1$ (A_8), the allocation of the *Span* between the two central offices Off_1 abd Off_2 (A_7), and the allocation of $Line_1$ (A_5) between the central office Off_1 to the *Start* point. However, according to the design in Figure 3(a), even if A_5 and A_7 are committed, A_8 cannot start until A_6 terminates. In contrast, the design in Figure 3(b) encourages the release of committed resources as soon as one of the subactivities commits. Hence, A_8 may access the objects released by A_5 and A_7 whenever A_5 and A_7 commit, rather than having to wait for the entire activity C to commit.

Using TAM, it is also possible to group the subactivities A_5, A_6, A_7, A_8 and A_9 by creating a new composite activity D:ALLOCATEFACILITIES as their parent activity. For example, this can be done through the combination of "*split-by-unnest*" operation over C, and "*join-by-nest*" operation over A_5, A_6, A_7, A_8 and A_9. The result is shown in Figure 3(c). Comparing with the initial design, this alternative design improves the cooperation among subactivities A_5, A_6, A_7, A_8 and A_9 and adds higher concurrency, because if activities A_5, A_7 commit earlier than A_6, then A_8 may access the objects delegated to D by A_5, A_7, without waiting for the termination of A_6. Similarly A_9 may access the objects delegated to D by A_6, A_7, without waiting for the termination of A_5. Generally speaking, split-

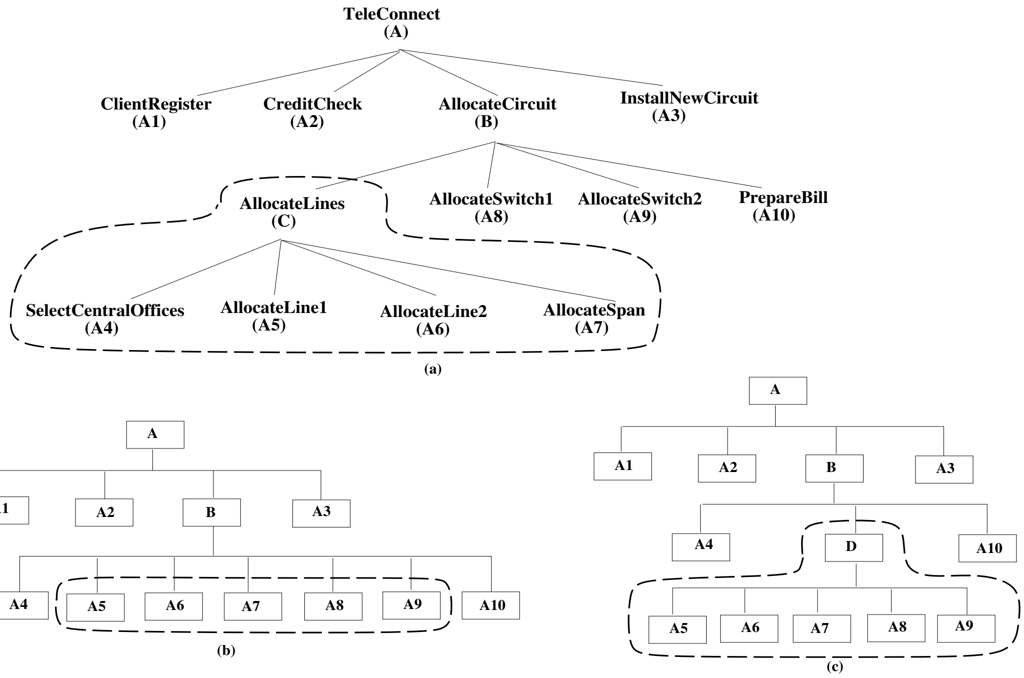

Figure 3: Alternative designs of the telephone connection application

activity divides an on-going activity into two or more activities that cooperate with one another, and each of the new activities is later committed or aborted independently of the others. Join-activity either merges or groups two or more activities that cooperate in terms of application-dependent contracts into a single activity as if they had always been part of the same activity, and all their work is now committed or aborted together.

3.2 Activity-Split and Activity-Join Operations

The *activity-split* operations divide an ongoing activity, say C, into two or more activities, say C_1, C_2, and C_3. We can also make C_1 to be the original activity. C_1, C_2, and C_3 may be independent, in which case they can commit or abort independently; or they may be dependent (say C_k is the parent activity of C_i, $i, k \in 1, 2, 3,\ i \neq k$), in which case C_i is *termination-sensitive* to C_k. When C_k aborts, if C_i is still *active* then C_i aborts; and if C_i is *commit*ted then C_i is compensated. Whether C_1, C_2, and C_3 are independent or dependent is primarily an application-specific design decision made by users. In case of elementary activities (of ECAT patterns), a system-default option can be provided in terms of the objects accessible to each of the split activities, for example, an elementary activity can be divided into two serializable transactions using the mechanisms defined in [21, 12]. Thus our discussion herein focuses on composite activities (of CACT patterns).

The inverse operation of split-activity, called *join-activity*, can combine results of several subactivities together and release them atomically. In particular, two or more ongoing activities can be merged into one, as if they had always been a single activity. This feature allows to hand over results of a user activity to a co-worker to integrate into his/her own ongoing task.

Two types of join-activity operations are currently supported in TAM. The operation "*join-by-nesting*" groups two or more activities by creating a new activity as their parent activity. The operation "*join-by-unnesting*" merges two or more activities into one single activity. To ensure the correctness criteria of join-activity operations, the merged or grouped activities must preserve all application-specific activity dependencies, such as subactivity execution rules, subactivity interleaving rules and state transition rules.

In summary, split-activity and join-activity can be combined in any formation. By allowing to release early-committed resources or transfer ownership of uncommitted resources, these dynamic restructuring operations bring a number of advantages, such as added concurrency, enhanced cooperation, and adaptive recovery, to the composable activity model for organizing cooperative activities in distributed and multi-user design and computing environment.

4 Related Work

Several activity models have been proposed [7, 6, 22] to support declarative specification of control flows within activities. Features of long running activities [7, 6] include an automatic compensation capability that offers some level of failure atomicity for the activity and the use of ECA-rules for monitoring activities. The cooperative model [22] achieves cooperation by controlled data exchange of the content of workspaces and controlled sharing of a common database among users. Although these models all provide a rich set of modeling constructs for capturing control flows within business activities, to the best of our knowledge, very few has provided a transactional support for split-activity and join-activity operations by ensuring the correctness properties of resulting activities in terms of the correctness specification of the original activities.

The development of TAM can be seen as an important extension to the activity composition model proposed in [14] by adding correctness (transactional) properties and incorporating dynamic activity restructuring operations into the activity execution. Our early proposal [14] focused more on the scheduling of message exchanges within an activity execution. However, no considerations were given for concurrent execution of activities and for studying the correctness properties of dynamic split-activity and join-activity operations.

A concrete implementation of TAM is being built on top of the two prototypes: the distributed information mediation system (DIOM) [15] and the Reflective Transaction Adapters [3], a toolkt for building the Reflective Transaction Framework on top of Transarc Encina, a commercial TP monitor. Of particular interest to TAM is the distributed query services in DIOM which can be adapted directly for transaction routing, decomposition, and distributed concurrency scheduling, and the support in the Reflective Transaction Framework for the implementation of many semantics-based concurrency control methods in online TP monitors.

Appropriate recovery of composite activities is a challenging research topic on which significant progress has been made. Workflow systems such as Exotica [18], for example, contain useful components for the recovery of elementary activities such as reliable/recoverable queues and a state machine interpreter. Flexible and modular implementation of coordination protocols [23] is another example of important components of a solution for complex activity recovery.

5 Conclusion

We have described a reflective framework for organizing and restructuring of complex activities. The framework consists of the Transactional Activity composition Model (TAM) and a family of activity-split and activity-join operations. The main contribution of the paper is a simple and effective facility to allow activity designers to specify the hierarchical organization of complex activities and the wide spectrum of interaction dependencies among concurrent activities declaratively and incrementally. TAM activity dependency constructs, such as **precede**, **enable**, **disable**, and **compatible**, present a clean separation of activity specification from their implementation to support reasoning about correctness of concurrent activities independently of their implementation. To demonstrate the expressive power and benefits of using our reflective framework for organizing and restructuring complex activities, we have outlined an application drawn from telecommunications.

References

[1] M. Ansari, L. Ness, M. Rusinkiewicz, and A. Sheth. Using flexible transactions to support multi-system telecommunication applications. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 65–76, 1992.

[2] F. Bancilhon, W. Kim, and H. Korth. A model for cad transactions. In *Proceeding of the 11th International Conference on Very Large Databases*, pages 25–33. Morgan Kauffman, 1985.

[3] R. Barga and C. Pu. A practical and modular implementation technique of extended transaction models. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, September 1995.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in database ststems*. Addison-Wesley, 1987.

[5] P. A. Bernstein, J. Rothnie, N. Goodman, and C. Papadimitriou. The concurrency control mechanism of sdd-1: A system for distributed databases (the full redundant case). *IEEE Trans. on Software Engineering*, 4(3), May 1978.

[6] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the ACM SIGMOD*, 1991.

[7] U. Dayal, M. Hsu, and R. Ladin. A transactional model for long-running activities. In *Proceedings of the 17th Very Large Databases*, pages 113–122, 1991.

[8] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[9] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. on Database Systems*, 8(3), June 1983.

[10] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and management of extended transactions in a programmable transaction environment. In *Proceedings of the 1994 IEEE Conference on Data Engineering*, pages 462–473, Feb 1994.

[11] D. Georgakopoulos, M. Rusinkiewicz, and W. Litwin. Chronological scheduling of transactions with temporal dependencies. *Very Large Database Journal*, January 1994.

[12] G. Kaiser and C. Pu. Dynamic restructuring of transactions. In A. Elmagarmid, editor, *Transaction Models for Advanced Applications*. Morgan Kaufmann, 1991.

[13] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA Workshop on Reflection and Meta-level Architectures*, 1992.
See `http://www.xerox.com/PARC/spl/eca/on.html` for updates.

[14] L. Liu and R. Meersman. Activity model: a declarative approach for capturing communication behavior in object-oriented databases. In *Proceeding of the 18th International Conference on Very Large Databases*, pages 481–493, Vancouver, Canada, 1992. Morgan Kauffman. The long version of this paper is published in ACM Transactions on Database Systems, Vol.21, No.3, 1996.

[15] L. Liu, C. Pu, and Y. Lee. An adaptive approach to query mediation across heterogeneous databases. In *Proceedings of the International Conference on Coopertive Information Systems*, Brussels, June 19-21 1996.

[16] L. Liu, C. Pu, and R. Meersman. The transactional activity model for organizing open-ended activities. Technical Report TR96-21, University of Alberta, Department of Computing Science, April 1996.

[17] N. Lynch. Multilevel atomicity: A new correctness criterion for database concurrency control. *ACM Trans. on Database Systems*, 8(4), December 1983.

[18] C. Mohan, G. Alonso, R. Gunthor, and M. Kamath. Exotica: A research prespective on workflow management systems. In *IEEE Bulletin of the Technical Committee on Data Engineering*, pages 19–26, March 1995, Vol.18, No.1.

[19] M. Nodine, S. Ramaswamy, and S. Zdonik. A co-operative transaction model for design databases. In *Elmagarmid [8]*, pages 53–85, Chapter 3, 1992.

[20] M. Nodine and S. Zdonik. Cooperative transaction hierarchies: a transaction model to support design applications. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 83–94, 1990.

[21] C. Pu, G. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 27–36, Los Angeles, August 1988.

[22] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wasch, and P.Muth. Towards a cooperative activity model - the coopertive activity model. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 194–205, 1995.

[23] T. Zhou, C. Pu, and L. Liu. Adaptable, efficient, and modular coordination of distributed extended transactions. In *Proceeding of the International Conference on Parallel and Distributed Databases*, 1996.