A Transactional Activity Model for Organizing Open-ended Cooperative 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

Robert Meersman
Dept. of Computer Science
Vrije University of Brussel
StarLab, Pleinlaan 2
1050 Brussels, Beligum
meersman@vub.ac.be

April 10, 1996

Abstract

A number of extended transaction models have been proposed to support information-intensive applications, such as CAD/CAM, distributed operating systems, and software development. Irrespective to how successful these extensions are in supporting the systems they intended for, these model can only capture a subset of the interactions required in any complex and distributed information systems. Moreover, extended transaction models that offer adequate correctness in one application may not ensure correctness in other applications, because cooperation restrictions desired by one application domain may not be required or may even be unacceptable by another.

We propose a transactional activity model TAM for specification and management of open-ended cooperative activity by promoting the separation of activity specifications from their implementation, to allow reasoning about transactional properties of open-ended activities independently from their transaction implementation techniques, and by developing mechanisms to facilitate the specification and reasoning of application-specific activity dependencies. TAM users may form new activity patterns by combining components of different existing patterns using activity pattern refinement and activity pattern composition mechanisms. A number of activity restructuring operations are introduced and incorporated into the TAM activity model. To guarantee the correctness of new activities generated by activity-split or activity-join operations, we identify the cases where the correctness is ensured and the cases where activity-split or activity-join are illegal due to the inconsistency incurred. We also formally define the concept of complete activity history, a valid prefix of activity history, and a merged activity history. Our algorithm for integrating two meragable activities ensures that a merged history from two correct histories is also correct with respect to the two input histories. The correctness reasoning capability is tested by studying the new properties of activity-split and the concurrent execution of activities by multi-users in the context of TAM.

Keywords: *Distributed and heterogeneous systems, open-ended activities, cooperative activities, extended transaction models.*

*Supported partially by NSERC grant OGP-0172859 and NSERC grant STR-0181014

†Supported partially by ARPA grant N00014-94-1-0845, NSF grant IRI-9510112, and grants from the Hewlett-Packard Company and Tektronix.

Contents

1 Introduction

Open-ended activities are characterized by long duration, flexible cooperation and dynamic development [30, 19]. The need to support open-ended activities emerges from the increasing demands of new and complex applications, such as CAD/CAM, automated office workflows information systems, software development environments, and distributed operating systems. These applications are typically distributed and object-based. Cooperation is promoted. Activities in the advanced applications tend to access many objects, involve lengthy computations, and often require application-specific cooperation among multiple subactivities in accomplishing a task. The subactivities may be executed by different servers on different nodes of a heterogeneous service network. Furthermore, construction of subactivities may evolve as the work progresses to improve the quality of cooperation and obtain a higher degree of concurrency. Traditional transactions provided in conventional database systems [18], although powerful, were assumed to be short-lived and were targeted for competitive environments where interactions are curtailed. Therefore, the need to capture open-ended and collaborative activities found in the new applications suggests the need for more cooperative models.

A number of *extended transaction models* (ETMs) [13, 15, 7, 24, 27, 32, 30, 12] have been proposed to support diversified new application requirements. For example, Nested Transactions [25] were proposed in the context of distributed systems to handle the problems of partial failure. Sagas [15] extend the traditional transaction model by including an automatic compensation capability within transactions to relax failure automicity. Cooperative Transactions [2], Split Transactions [30], and Transaction Groups [27] were proposed for capturing the interactions required in advanced applications. Most of these ETMs were targeted at a particular domain of applications, and can only capture a subset of the spectrum of interactions found in any complex information systems [7, 23]. Furthermore, ETMs that offer adequate correctness in one application may not ensure correctness in other applications. Some cooperation restrictions desired by one application domain may not be required or even unacceptable by another. For example, nested transactions do not allow sharing of uncommitted data. This requirement is often critical in CAD/CAM applications and interactive programming environment. Sagas cannot guarantee database correctness crucial to many banking applications. Unlike nested transactions, cooperative transactions [2, 30, 27] allow cooperation among siblings. However, the interactions among siblings are either restricted to leaf node transactions in order to support serializable split-transactions or limited to static and one-shot design of transaction groups, making it difficult to introduce added concurrency and improve the cooperation through releasing some early committed resources or transferring ownership of uncommitted resources, once the transaction groups are defined. For these reasons, a distributed and interoperable object management system should provide an adaptive facility that supports the definition and construction of specific ETMs corresponding to various application requirements, rather than a single built-in ETM. It should also be possible to enforce these specifications efficiently for concurrency control. Put differently, a transactional activity specification language must provide modeling primitives, similar to those in the ACTA framework [7, 8, 9], to allow the (explicit) specification of application-specific activity dependencies, while satisfying the basic requirement that it must not require complete histories to reason about the correctness of concurrent execution of activities.

We develop a transactional activity model TAM for specification and management of open-ended cooperative activities. TAM provides modeling primitives that allows transaction model designers to create implementation-independent specification of complex transactional activities. We provide a number of facilities to allow explicit specification of activity dependencies between subactivities, between subactivities of siblings, and with respect to single or multiple threads of object flows. Communication among subactivities of an activity is carried out via parameter passing. Semantics of interactions between

activities are expressed in terms of their effects on each other and on objects they access. The effects of one activity on other activities include *not only* those on the *abort* of other activities and those on the *commit* of other activities, *but also* those on the execution precedence and interleaving with respect to other activities. The effects of an activity on the objects that it accesses are captured by means of access sets of activities and the concept of delegation [7, 31]. TAM also supports the behavioral refinement and behavioral composition of activity patterns to increase the flexibility and reusability of activity patterns. The separation of activity specification from implementation is a key concept which allows reasoning about concurrency and recovery properties of TAM activities independently from their transaction implementation mechanisms [3, 16]. The reasoning capability has also been tested by using the TAM to study the new properties of activity split in the context of TAM.

The development of TAM can be seen as a continuation of the activity model proposed in [21] by adding transactional properties and by incorporating dynamic restructuring operations [30, 19] into the activity execution. For example, In our early proposal [21] there was no consideration of new criteria for fault tolerance, failure recovery, and correct cooperation among concurrent users. In TAM, we precisely define the semantics of compensation and exception handling of activities. Instead of concurrency atomicity and failure atomicity, we guarantee that open-ended activities are executed in accordance with the application-specific activity dependencies defined by the activity/transaction designer and terminate only in the legal termination states. Upon failure, we allow activities to be aborted or compensated or to handle the failure exception by executing an alternative subactivity. Another feature supported by TAM is that activity patterns can be defined statically and modified dynamically through the invocation of system-supplied activity restructuring operations in the progress of activities.

Our transactional activity model is especially useful for distributed systems because it supports intra-activity parallelism by allowing an activity to split into nested activities that can execute concurrently. TAM can be layered on more than one transaction models that support atomic transactions. This is particularly useful in heterogeneous environment where different servers may support different transaction models. The support for specific ETMs in TAM involves translating specifications to instructions carried out by the local programmable transaction management systems. Thus specification of popular ETMs (such as nested model, sagas) can be provided in TAM as templates. Activity designers may combine components of different ETMs, by means of activity pattern refinement and activity pattern composition, to form new ETMs.

In addition to our early proposal [21], a few activity models have been proposed [11, 10, 31] to support declarative specification of control flows within activities. Features of long running activities [11, 10] 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 [31] achieves cooperation by controlled exchange and synchronization of the content of workspaces among users. However, these models provide no support of high-level activity abstraction mechanisms for behavioral refinement and behavioral composition of activity patterns. The correctness criteria for concurrent execution of activities have not been defined and explored formally. Dynamic restructuring of activities has not been addressed and studied in the context of activities of arbitrary nested hierarchical structure. We argue that in an open environment such as distributed information systems, activity abstraction mechanisms and activity restructuring facilities are useful not only for defining new patterns (templates) of transactional activities in terms of existing ones and for adding concurrency and improving cooperation, but also for making the TAM model scalable to the dynamic evolution of activity development in the presence of transactional requirement changes, and to the increasing number of information servers in high-speed networks.

In Section 2 we introduce the TAM activity model, including the formal definition of simple and com-

posite activity patterns, and the modeling primitives for specification of user-defined activity execution dependencies, interleaving dependencies, and state transition dependencies. Section 3 discusses issues related to ensuring correctness criteria. Section 4 introduces a number of activity restructuring operations for dynamic split and join of activities. An application scenario in telecommunication domain is used in Section 5 to illustrate the concepts discussed in this paper.

2 The Transactional Activity Model TAM

2.1 Hierarchical and Dynamic Organization of Activities

In TAM *open-ended* activities are specified by activity templates or so called parameterized *activity patterns*. An activity pattern describes concrete activities occurring in a particular organization. Each concrete activity is described by a single activity pattern and can be seen as an instantiation of the activity pattern.

An activity (say P) may consist of several child activities (or so called *subactivities*). A child activity may in turn have subactivities as its own children and so on. All the direct and indirect subactivities of P are also called *constituent* activities of P. The TAM model allows arbitrary nesting of activities since it is generally not possible to determine a priori the maximum nesting an application task may need. We refer to the hierarchical organization of activities of as *activity hierarchy*.

To facilitate the abstract specification of the hierarchical organization of *open-ended* activities, TAM distinguishes two types of activity patterns: *simple activity patterns* (SACT) and *composite activity patterns* (CACT). A simple activity of patterns (SACT) is a program that issues a stream of messages to access the underlying database. They are leaves of the activity hierarchy. A composite activity pattern (CACT) represents activities that consist of a number of subactivities cooperating to accomplish a specific task. Composite activity patterns are the internal nodes and the *root* of the activity hierarchy.

2.1.1 A Running Example

Consider a complex activity TeleConnect that performs installing and billing of telephone connection for a telecommunication company [1]. Suppose the activity A:TeleConnect consists of four subactivities A_1:ClientRegister, A_2:CreditCheck, B:AllocateCircuit, and A_3:InstallNewCircuit. 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. Activity A_2 evaluates the credit history of the client and pass a service order to the activity B:AllocateCircuit. Activity B attempts to provide a connection by allocating existing resources such as selecting nearest central offices, and using existing lines and slots in switches. If B succeeds, the cost of connection is minimal. Thus, activity A_3 is designed to performs an alternative task that involves physical installation of new facilities in the event of failure of activity B. Figure 1(a) presents a possible initial task breakdown of the example activity TeleConnect. A_i $(i = 1, ..., 10)$ are simple activity patterns, and A:TeleConnect, B:AllocateCircuit and C:SelectConnections are composite activities.

[1] A simpler version of designing telephone circuit between two points is described in [1, 16].

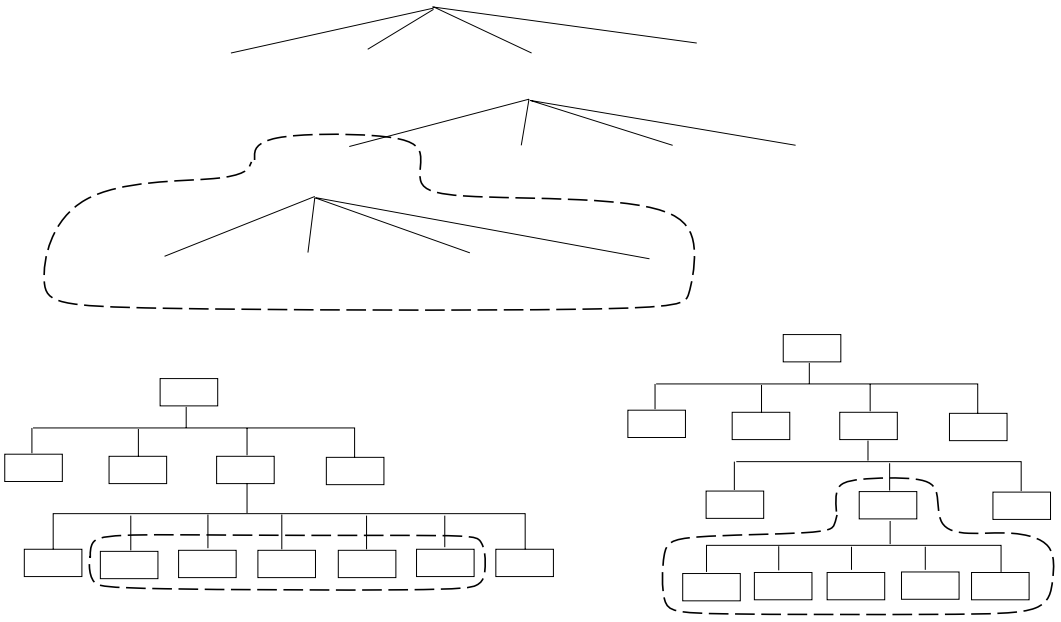

Figure 1: Alternative designs of the telephone connection application

2.1.2 Dynamic Split and Join of Activities

Often the structure of most design tasks may evolve as the work progresses, a dynamic hierarchy is essential for such *open-ended* activities. In TAM, we allow subactivities to be added or removed from an activity hierarchy as the need arises, because we can not always foresee the most effective clustering structure of activities at the stage of initial activity specification. For instance, the designer of the activity AllocateCircuit may decide later to unnest the AllocateLines activity, through "*split-by-unnest*" operation, as shown in Figure 1(b), in order to add concurrency and to improve cooperation among subactivities A_5, A_6, ..., A_9. This is because, in the initial design, activity A_6 has nothing related to the allocation of $Switch_1$ between $Line_1$ and the *Span* between $Line_1$ and the central office to *Start* point. However, according to the design in Figure 1(a), even if A_5 and A_7 are committed, A_8 cannot start until A_6 terminates. Whereas the design in Figure 1(b) encourages to release the committed resources as soon as one of the subactivities commits. Hence, A_8, A_9 may access the objects whenever A_5 and A_7 commit, rather than waiting for the entire activity C to commit.

It is also possible if the designer decides 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, 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, as shown in Figure 1(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. Similar situation holds between A_6, A_7, A_9 and A_5. Generally speaking, split-activity divides an on-going activity into two or more activities that cooperate with respect to 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.

Organizing *open-end* activities hierarchically and dynamically has many advantages. First of all, such a nesting hierarchy reflects a natural task breakdown common in most cooperative working environments. A complex tasks can be designed by top-down or bottom-up or mixed strategies through activity abstractions. Details of subtasks can be isolated from its parent and siblings, facilitating activity abstraction and overall task management. Secondly, the hierarchical structure of activities allows subactivities of the same parent activity to run concurrently, shortening the perceived execution time of the activity. Thirdly, comparing with a flat organization where failure of a subactivity would impact all work, related or unrelated, the multi-layered nesting structure of activities enables to keep failure recovery.under control. When a subactivity fails, we only have to undo its effects. This effectively shields its siblings. Last but not least, dynamic split and join are important facilities for organizing *open-ended* activities hierarchically. Split-activity is useful for committing some work early or dividing on-going work among co-workers. Join-activities allow to hand over results to co-workers to integrate into his/her own on-going work [30, 19].

2.2 Activity Patterns: Formal Definition

Given a universe of discourse (UoD), the activity model of this UoD consists of a non-empty set of activity patterns, describing Application-specific tasks and their communication interfaces and cooperation contracts.

An activity pattern describes the communication protocol of a group of user activities that hold the similar hierarchical structure and obey the similar cooperation contracts. As we mentioned earlier, TAM distinguishes two types of activity patterns: *simple activity patterns* (SACT) and *composite activity patterns* (CACT). A simple activity of patterns (SACT) is a program that issues a stream of messages to access the underlying database. A composite activity pattern (CACT) consists of a number of subactivities cooperating to accomplish a task. We identify the following information necessary for specification of a simple activity pattern:

- The type name of each participating object in an activity (so called an agent of the activity).
- The list of input and output parameters.
- A set of messages used by each agent object in participation of an activity.
- The logic of message exchanges among agent objects, including the local constraints on the set of permissible messages of each agent and the global constraints that describe the kind of mutual obligations the participating objects should follow.
- The *pre-* and *post-*conditions as well as the *initiation* condition of an activity execution. For example, the precondition may describe what sorts of relationships must hold between agent objects. The initiation condition describes when an activity can be invoked. The postcondition specifies the post-effect that an activity might have and the consistency that should be maintained.
- The activity dependencies that identify the correctness constraints of synchronization of executions of activities.

Definition 1 (simple activity pattern)
A simple activity pattern α is described by an octuple (N, AGT,PARA, GIC, PRE, INIT, POST, EXCE)

N: *is the name of activity pattern α.*

AGT *is a nonempty set of agent objects, each agent is described by a triple (T,Msg(α,T),LIC(α,T))*
- *T* *is the type name of an agent involved in the activity pattern a and called agent type,*
- *Msg(α,T)* *is a nonempty set of messages supporting the role of type T being an agent of α, type T being an agent in the activity pattern α,*
- *LIC(α,T)* *denote the conjunction of all the local constraints associated with the agent of type T in activity pattern α, which defines the exchange ordering of the messages in Msg(α,T).*

PARA *is a set of input/output parameters used in an activity of pattern α, which can be names of attributive properties of the participating agents.*

GIC *denote the conjunction of all the global constraints specified in activity pattern α, which express the interleaving rules of the messages between different agent objects.*

PRE *is a set of preconditions that hold at the start of an activity of pattern α (the sufficient conditions).*

INIT *be a conjunction of all initiation (triggering) conditions which must hold in order to initiate an activity of pattern α*

POST *is a set of initiation or triggering conditions that must hold in order to initiate an activity of pattern α (the necessary conditions).* □

When all the messages involved in an activity are executed in a sequential order, the activity corresponds to the transaction in the traditional database management system. Since the concept and specification of simple activities have been discussed in our early publication [21], we omit the further illustration and examples here.

A composite activity pattern presents a structured set of subactivities, whose interactions are structured to reflect the underlying hierarchical decomposition of the task to be accomplished. In addition to the information specified in the definition of a simple activity pattern, a composite activity pattern, say α, should also specify:

- the activity execution dependencies and the activity state transition dependencies among its constituent activities.
- the behavioral composition constraints that the activity pattern α and its constituent activity patterns $\beta_1, ..., \beta_n$ should satisfy.

These additional properties are critical for distinguishing the behavioral composition relationship between activities and the invocation control relationship between activities.

Definition 2 (composite activity pattern)
Let Agents(α) be the set of agents in activity pattern α, and Msg(T,α), PARA(α), LIC(T,α), GIC(α), PRE(α), INIT(α), POST(α) be defined as in Definition 1. Let EXERULES(α) be a set of activity execution and interleaving dependencies rules of the activity pattern α. Let STRULES(α) be a set of user-defined activity state transition rules that augment the system-defined default conditions for activity state transition. An activity pattern α is a **composite** *activity pattern, if and only if (iff) there exist activity patterns $\beta_1, ..., \beta_n$ ($n \geq 1$) such that activity pattern α is a behavioral aggregation (composition) of $\beta_1, ..., \beta_n$ satisfying the following conditions:*

1. $\forall S \in Agents(\beta_i)$ $(1 \leq i \leq n)$, $\exists T \in Agents(\alpha)$ *such that*
   $(S = T \vee ComponentOf(S,T)) \wedge Msg(T,\alpha) \supseteq Msg(S,\beta_i) \wedge LIC(T,\alpha) \supseteq LIC(S,\beta_i)$.

2. $PARA(\alpha) \supseteq PARA(\beta_i)$.

3. $GIC(\alpha) \Rightarrow GIC(\beta_i)$.

4. $PRE(\alpha) \supseteq PRE(\beta_i)$.

5. $POST(\alpha) \supseteq POST(\beta_i)$.

6. $EXERULES(\alpha) \supseteq EXERULES(\beta_i)$.

7. $STRULES(\alpha) \supseteq STRULES(\beta_i)$.

8. *The relationship between the initiation condition of* α *and the initiation condition of* β_i *satisfies the following condition:*

   (a) *If activity patterns* $\beta_1, ..., \beta_n$ $(n \geq 1)$ *are executed in a sequential order, i.e.,* β_1 *before* β_1... *before* β_n, *then we have*
   $(\Box...(\Box(\Box INIT(\alpha) \Rightarrow \Diamond INIT(\beta_1)) \Diamond INIT(\beta_2)) \Diamond ... \Diamond INIT(\beta_n))$.

   (b) *If* α *is composed by a selection of activity patterns* $\beta_1, ..., \beta_n$ $(n \geq 1)$, *i.e., the execution of* $\beta_1, ..., \beta_n$ *is going to be selectively synchronized, then we have*
   $\Box INIT(\alpha) \Rightarrow \Diamond (INIT(\beta_1) \vee INIT(\beta_2) \vee ... \vee INIT(\beta_n))$.

   (c) *If* α *is is composed by a concurrent synchronization of activity patterns* $\beta_1, ..., \beta_n$ $(n \geq 1)$, *i.e., the activities* $\beta_1, ..., \beta_n$ *can be executed concurrently, then we have*
   $\Box INIT(\alpha) \Rightarrow \Diamond (INIT(\beta_1) \wedge INIT(\beta_2) \wedge ... \wedge INIT(\beta_n))$.

   (d) *If* α *includes a repeated synchronization of an activity pattern* β_i $(1 \leq i \leq n)$, *i.e., the execution of activity* β_i *can be repeated zero or more times within the execution of an activity of pattern* α, *then we have* $\Box INIT(a) \Rightarrow \Diamond INIT(\beta_i)$.

We refer to β_i $(i \in \{1, ..., n\})$ *as constituent activity (or subactivity) pattern of* α. □

Condition (1) states that if activity pattern α is a behavioral composition of activity patterns β_i $(1 \leq i \leq n)$, then for any type (say S) in Agents(β_i), there is a corresponding type T in Agents(α) such that either T and S are the same or the predicate ComponentOf(S,T) holds. In the later case, all messages in the message set Msg(S,β_i) are also messages in the message set Msg(T,α). Note that it is also possible for a composite activity pattern to have additional agent types in its agent set. This means that when a composite activity pattern involves n agents, some of its constituent activity patterns may have less than n agents. Semantics of Condition (2) to Condition (7) are straightforward. Condition (8) means that, from now on, if the conjunction of all initiation conditions stated in a composite activity pattern α holds, then for each constituent activity β_i $(1 \leq i \leq n)$, there should be a time point in the future such that the conjunction of all messages specified in β_i will be verified.

The temporal modal operator $\Box$ refers to "*always in the future*" and the modal operator $\Diamond$ refers to "*sometimes in the future*". As this paper focuses only on the specification of various application-dependent activity dependencies and the reasoning about correctness of concurrent activity executions, readers who are interested in examples and further details on the usage of temporal modal operators may refer to [21].

Proposition 1 *Let the relation predicate* BehavioralAgg(α,β) *denote that the activity pattern* α *has a behavioral aggregation relationship with the activity pattern* β. *Let* Σ_{ACT} *be a set of activity patterns with respect to an application domain, and* $\mathbf{L} = (BehavioralAgg,\ \Sigma_{ACT})$. $\mathbf{L}$ *is* **a partial order** *with* Σ_{ACT} *as the domain.*

Proof
By the definition of partial order [4], all we need to prove is that the relation BEHAVIORALAGG is an irreflexive, transitive binary relation on Σ_{ACT}. Let $\alpha == \beta$ denote that the two activity patterns are identical. we want to prove that, $\forall \alpha, \beta, \gamma \in \Sigma_{ACT}$, the following assertions are true.

$$BehavioralAgg(\alpha,\beta) \wedge BehavioralAgg(\beta,\gamma) \Rightarrow BehavioralAgg(\alpha,\gamma).$$

$$BehavioralAgg(\alpha,\beta) \wedge BehavioralAgg(\beta,\alpha) \Rightarrow \alpha == \beta.$$

By Definition 2, the laws of set inclusion, and logical implication, the proof is straightforward. □

This proposition states that a composite activity pattern can be defined by recursive application of the activity aggregation relation to the existing activity patterns. We call the activity aggregation hierarchy anchored at α the *activity hierarchy* of α. We call a constituent activity that is directly related to α through the *BehavioralAgg* relation the *child* activity of α.

In the rest of the paper we discuss activity patterns primarily in terms of the input and output parameters, the set of constituent activities, the set of activity execution dependencies defined by *EXERULES*, and the set of activity state transition rules specified by *STRULES*. This is partly because in this paper we concentrate mainly on the transactional properties and the new results of the activity model TAM, rather than the complete syntax and semantic development of the activity model, and partly because TAM is developed based on our earlier proposal [21] where specification of simple activity patterns was studied extensively. For concrete syntax and examples of the simple activity specification, Readers may refer to [20, 21] for detail.

2.3 Concurrent Execution of Activities: Correctness Specification

Temporal precedence and cooperative constraints are two typical correctness specification for open-ended activities. In contrast to traditional serializability theory [4], temporal precedence correctness criteria consider histories *correct* only if they are equivalent to one specific serial history [17]. Cooperative correctness criteria [5, 14, 22, 26, 16] use less restrictive notions of conflicts that take into account application-dependent semantics, and allow *compatible* transactions or activities to cooperate, e.g., to repetitively read and write specific objects without restrictions.

Two activities are defined to be *compatible* if and only if their execution order does not matter in terms of the semantic specification of the application. When the execution order of two subactivities within a composite activity P is important to the correctness of concurrent executions of P, we call these two subactivities *incompatible* w.r.t. P. Subactivities are compatible if they commute [12]. However, non-commuting subactivities may still be compatible, depending on the application-specific semantics. 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, the correct interaction among subactivities of a composite activity is derived from a set of user-defined activity state transition rules and activity execution dependency constraints. These application-specific dependency rules can be viewed as integrity constraints on the execution of complex activities.

They are utilized to ensure that the subactivities interact only in the ways *allowable* by its correctness specification such that execution of the composite activity guarantees to leave the database(s) in a correct state.

2.3.1 Activity State Transition Dependencies

In TAM each activity has a set of observable states S and a set of possible state transitions φ: S$\longrightarrow$S, where S = {*active, commit, abort, done, compensate*}. The transitions between the states of an activity can be represented by a finite state automaton and its transition graph. When an activity T is activated, it enters the state *active*. The *active* state of T is changed to *commit* if T commits, and to *abort* if T or its parent abort. After T is committed, if its parent aborts, then its state is changed to *compensate*. If its parent commits, then its state is transformed to *done*. These rules only presents the system-defined default state transitions of an activity. The set of state dependencies of a composite activity includes also the state dependencies between the composite activity P and its constituent activities, say $A_1, ..., A_n$. In TAM we define that an activity P commits only if all its constituent activities A_i $(i = 1, ..., n)$ legally terminate (i.e., commit or abort). In the other words, an activity P may commit even if one of its constituent activities is aborted. Thus, to be able to support the user-specific *abort-dependency* requirement, we allow the programmers to define whether an activity is *abort-critical* to its parent or not. The abort of an *abort-critical* activity will cause its parent to abort. For example, in the telecommunication application scenario, if the design of activity AllocateCircuit specifies that the abort of the subactivity SelectCentralOffices causes the abort of AllocateCircuit, then we call activity SelectCentralOffices *abort-critical* to activity AllocateCircuit. Let *Children*(P) denote the set of children activities of P. The state dependencies between P and its constituent activities can be defined in terms of enabling condition and the effect of state transition as shown in Figure 2:

State	**Enabling Condition**	**Post-Effect**
$active(P)$	$\exists A_i \in Children(P)\ active(A_i)$	none
$commit(P)$	$\forall A_i \in Children(P)\ (commit(A_i) \vee abort(A_i))$	none
$abort(P)$	$(\exists A_i \in AbortCritical(P)\ abort(A_i))$ $\vee\ (\forall A_i \in Children(P)\ abort(A_i))$	$\forall A_i \in Children(P)\ (active(A_i) \longrightarrow abort(A_i)$ $\vee\ commit(A_i) \longrightarrow compensate(A_i))$
$Done(P)$	If P is a root activity then $commit(P) \longrightarrow done(P)$, and if P is a non-root activity then $done(P)$ only if P's parent commits	none
$compensate(P)$	P's parent aborts after P is committed	none

Figure 2: State dependencies between an activity and its constituent activities

Using TAM activity specification language, users may override the default activity state transition dependencies by explicitly specifying the augmented state transition conditions in the activity pattern definition.

2.3.2 Specification of Activity Dependencies

Specification of *open-ended* activities is based on the observation that complex activities consists of a set of constituent activities and a set of activity dependencies that define the cooperation contracts (such as access rules to shared objects) between the *root* activity and its constituent activities and between constituent activities. Each constituent activity is of either a simple activity pattern (SACT) or a complex activity pattern (CACT). Activity dependencies represent the application-

dependent inter-task dependency constraints that are required to satisfy in the breakdown of a complex task. For example, consider the telecommunication application scenario given in Section 2.1.1. The activity A:TeleConnect consists of four subactivities A_1:ClientRegister, A_2:CreditCheck, B:AllocateCircuit, and A_3:InstallNewCircuit. A is executed when the telephone company receives a customer's request for telephone service installation. Based on the given application-dependent requirements, B and A_3 perform two alternative line allocation tasks. Either of them will result in a completed circuit. Thus, only one should be allowed to complete. We may refer to A_3 as a *contingency* activity [6, 7] with respect to B.

TAM provides four constructs for activity dependency specification: **precede, enable, disable, compatible**. The semantics of each construct is described in Figure 3. The construct **precede** is used to represent the *commit-active* state dependency. The construct **enable** and **disable** are utilized to specify the enabling and disabling dependencies between activities. The construct **compatible** describes the compatibility of activities A_1 and A_2.

Construct	Usage	Synopsis
precede	A_1 **precede** A_2	$commit(A_1) \longrightarrow active(A_2)$
enable	condition(A_1) **enable** A_2 condition(A_1) **enable** condition(A_2)	$condition(A_1)$='`true`' $\longrightarrow$ $active(A_2)$ If $condition(A_1)$='`true`' then condition(A_2) can be '`true`'
disable	condition(A_1) **disable** A_2 condition(A_1) **disable** condition(A_2)	$condition(A_1)$='`true`' $\longrightarrow$ $abort(A_2)$ If $condition(A_1)$='`true`' then $condition(A_2)$ cannot be '`true`'
compatible	**compatible**(A_1,A_2)	if *true* A_1 and A_2 can be executed in parallel, and if *false* the order of A_1 and A_2 is important

Figure 3: Constructs for activity dependency specification in TAM

Consider the telecommunication example. The following describes the application-dependent activity dependencies between AllocateCircuit and InstallNewCircuit:

(1) B, A_3 cannot begin (be *active*) before A_1 commits. I.e., A_1 **precede** B, A_3.
(2) A_3 cannot commit before B aborts. I.e., $abort(B)$ **enable** $commit(A_3)$.
(3) A_3 must abort if B commits. I.e., $commit(B)$ **disable** A_3.
(4) A_3 cannot begin after B has committed. I.e., $commit(B)$ **disable** $active(A_3)$.

Note that dependencies (3) and (4) imply "A_3 cannot commit after B commits", i.e., "$commit(B)$ **disable** $commit(A_3)$". However, this dependency cannot replace dependencies (3) and (4), because it allows A_3 to continue its execution, holding resources or performing unnecessary computations after B has committed.

2.3.3 Activity Execution Dependencies

Activity execution dependencies refer to both the temporal execution precedence of activities and the interleaving rules of activities. In TAM, both forward and backward activity execution dependencies can be specified by the designers. Forward execution dependencies control the routine execution of an activity, govern the execution sequence among subactivities of the same parent activity, and discipline the cooperation (synchronization constraints) of concurrent users involved in the same activity history (see Section 3.1 for detail). Backward execution dependencies govern the rollback of the activity, if it is not finished yet and need to be aborted. The rollback of committed activities is performed using compensation activities in the inverse order.

When a composite activity has a deeply-nested structure of subactivities that are related directly or indirectly with one another, it is natural to guide the activity designers to specify the application-specific execution dependency constraints by allowing them to concentrate on those activities in the hierarchy, which are directly related with each other, and let the system to reason about the dependency closure. Thus, we explicitly distinguish two types of of execution dependencies among constituent activities of activity P in the activity specification of P: **Subactivity execution rules** and **Subactivity Interleaving rules**. The former allows the designers to focus more on the execution order of subactivities at the same abstraction layer. The latter encourages the designers to specify the application-specific execution dependencies between children activities of siblings and between any subactivity and children of its siblings. For any given activity pattern α, the complete set of execution rules and interleaving rules of α can be computed easily according to Definition 2 and Proposition 1.

1. Subactivity Execution Rules

Activity execution rules are application-dependent semantic constraints on the occurrence of a subactivity execution and the temporal precedence of execution of subactivities with respect to the same thread of object-flow. In general, the execution rules can be defined in terms of various types of synchronization constraints, such as

- *Temporal precedence* of the execution of constituent activities. That is, subactivities $A_1, A_2, ..., A_n$ ($n > 1$) are executed in a pre-defined sequential order, denoted by A_1 **precede** A_2, ..., A_{n-1} **precede** A_n.

- *Selective execution* of a set of constituent activities in terms of output values of some other subactivity.

  Example: If the activity JOURNEY-CONTROL(pl) returns a control status '*emergency*', then instead of executing subactivity LANDING(pl, ct_2), the subactivity EMERGENCY-LANDING(pl, ct_2) shall be executed. We denote the selective execution semantics by means of the following three execution rules:

  (1) JOURNEY-CONTROL(pl) **precede** [LANDING(pl, ct_2), EMERGENCY-LANDING(pl, ct_3)].
  (2) *control-status*(pl)='*emergency*' **enable** EMERGENCY-LANDING(pl, ct_3)
  (3) *control-status*(pl)='*ok*' **enable** LANDING(pl, ct_2)

- *Conditional execution* of constituent activities in terms of external events.

  Example: Consider the two execution constraints: (1)"Subactivity A_1 repeats three times if the event E_1 occurs", and (2)"subactivity A_2 starts after 6:00pm". Let the predicate *Occurs*(E) denote the occurrance of an external event E and the predicate *Repeat*(A, n) denote the number n of times that activity A repeats. We express these two execution constraints as follows:

  Occurs(E_1)='`true`' **enable** *Repeat*($A_1, 3$).

  Occurs('`6am`') **enable** A_2.

Any combination of the above types is possible. We will discuss how the activity execution rules are used to reason about correctness of the activity execution, and correctness of the cooperation among concurrent executions of activities by multiple users in Section 3 and Section 5.

2. Subactivity Interleaving Rules

A set of subactivity interleaving rules defined in an activity specification (say α) describes (1)the execution dependencies between subactivities of a child activity and subactivities of its siblings; (2)the execution dependencies between subactivities of sibling activities, *and* (3)the compatible executions of the same activity w.r.t. different threads of object-flows or the compatible executions of dependent sibling activities w.r.t. different threads of object-flows.

Interleaving rules of the first two types may be derived by logic deduction from the set of subactivity execution rules and the nested structure of objects. However, the third type of interleaving rules (if any) has to be explicitly specified by the designers or users.

Consider a simple application in authoring environment, which requires cooperative editing of design documents based on annotation. Assume that EDIT(d:Document) and ANNOTATE(d:Document) are two constituent activities of activity DESIGNDOCUMENT. EDIT(d:Document) consists of m_d occurrances of activity CHPEDIT(p:Chapter). m_d is a number of chapters of the document d. ANNOTATE(d:Document) consists of m_d occurrances of CHPANNOTATE(p:Chapter). Two execution dependencies are identified by the designer of the activity DESIGNDOCUMENT: (1)Editing a document is based on the annotation previously made. (2)Two users may edit or annotate a document concurrently as long as they edit different chapters. We specify these application-dependent activity dependencies as follows:

1. (activity execution rule)
   $(\forall d \in$Document$)($ANNOTATE(d) **precede** EDIT$(d))$.

2. (activity interleaving rule)
   $(\forall d \in$Document$)(\forall chp_i, chp_j \in$Chapters$(d))$ $(chp_i \neq chp_j$
   $\wedge$ **compatible**(CHPEDIT(chp_i),CHPEDIT(chp_j)),
   $\wedge$ **compatible**(CHPEDIT(chp_i),CHPANNOTATE(chp_j)),
   $\wedge$ **compatible**(CHPANNOTATE(chp_i),CHPANNOTATE(chp_j)).

By means of the execution rule explicitly defined in the DESIGNDOCUMENT, and the knowledge that a document consists of a number of chapters, the following interleaving rule can be derived automatically:

$(\forall d \in$Document$)(\forall chp_i \in$Chapters$(d))$ CHPANNOTATE(chp_i) **precede** CHPEDIT(chp_i) $(1 \leq i \leq m_d)$.

Based on the execution rule and the interleaving rules of DESIGNDOCUMENT given above, we can obtain the following compatibility table with respect to subactivities E_i:CHPEDIT(chp_i), E_k:CHPEDIT(chp_k), A_i:CHPANNOTATE(chp_i), and A_k:CHPANNOTATE(chp_k). The symbol "Y" denotes that the two activities are compatible and the symbol "N" denotes that the concurrent execution of two activities are conflict and thus should be executed in a serializable order. In the other words, this compatibility table specifies the interleavings that are forbidden in the concurrent execution of subactivities of DESIGNDOCUMENT.

	E_i	E_k	A_i	A_k
E_i	N	Y	N	Y
E_k	Y	N	Y	N
A_i	Y	Y	N	Y
A_k	Y	Y	Y	N

Figure 4: The compatibility table for cooperating subactivities: an example

Assume, two users start editing the same version of the document but user A starts with ChpEdit(chp_i) and user B starts with ChpEdit(chp_k). According to the compatibility table in Figure 4, these two subactivities are compatible with each other. Thus when user B delegates her work to user A, user A can directly use the result that user B obtained before the delegation without worrying about whether the data obtained from user A is consistent with her own update.

Alternatively, assume, two users start editing the same version of the document, one starts by executing ChpEdit(chp_i) and one by annotating the same chapter ChpAnnotate(chp_i). According to the compatibility table above, these two user activities are incompatible, i.e., the concurrent execution of ChpEdit(chp_i) and ChpAnnotate(chp_i) will result in conflict. Therefore, if $User_1$ delegates her work to $User_2$ then $User_2$ has to redo the editing according to $User_1$'s annotation.

2.3.4 Object Replication in Activity Hierarchy

Another issue related to concurrent execution of activities is the issue of object copies at different points in the activity hierarchy [26]. In TAM the *multiple-object-replication* approach is used. Each composite activity has a local set of object versions. Thus, there may be many 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 more recent. With multiple object replications, a notion of visibility is defined naturally according to the hierarchy. That is, the version of an object in an activity is accessible to any of its descendant activities in the hierarchy. Put differently, an object in an activity's access set is copied automatically into a child activity's access set when the child activity initiates an access to that object. A new version of the object is written back to the parent activity's access set when the subactivity that initiates the access to the object comes to a *breakpoint* or commits. A *breakpoint* [5, 14] of an activity represents a point in its execution at which other activities can interleave. For example, say the composite activity DocumentDesign has made a copy of the Document object d. If the subactivity Annotate(d) has made a copy of it and modified the copy, then the new version is the one that the subactivity Edit(d) would read. In TAM each composite activity determines how resilient its object copies are to the various types of failures. The *Root* activity at the top of the activity hierarchy contains the most stable version of each object, and makes the guarantees that it can recover its copies of objects in the event of system failure.

An object in an activity's access set is copied automatically into a child activity's access set when the child activity initiates an access to that object. A new version of the object is written back to the parent activity's access set when the subactivity that initiates the access to the object comes to a *checkpoint* [26, 31] or commits.

Let A_P be a composite activity and A_C be a child activity of A_P. The relationship between activity A_P and its children, say A_C, can be described by the following properties:

- **Termination**: A_P commits only after A_C legally terminates.
- **Visibility**: A_C has access to all objects that A_P can access, i.e., it can read objects that A_P has modified.

In contrast to the application-dependent activity dependencies, these two types of dependencies are defined as system defaults. Users may define some mission-critical dependencies for termination and visibility to override these system defaults. In current development of TAM, overriding is only supported

for termination dependencies. We allow users to define application-specific abort-dependencies as the state transition rules in an activity specification.

Unlike in the conventional nested transaction model [7, 8], the *commit* of a subactivity in TAM is independent of the *commit* of its parent activity. Therefore, if an activity aborts, then all its *active* children are aborted; and *commit*ted children, however, are compensated for. We call this property *abort-sensitive* dependency between an activity A_C and its parent activity A_P, denoted by $A_P \rightsquigarrow A_C$. This abort-sensitive dependency prohibits a child activity instance from having more than one parent, ensuring the hierarchically nested structure of *active* activities. Put differently, there exists no such activities A, B, C in an activity hierarchy that $A \rightsquigarrow B \wedge C \rightsquigarrow B$ holds. When the abort of all active subactivities of the activity is completed, the compensation for committed subactivities is performed by executing the corresponding compensations in an order that is the reverse of the orignal order.

2.4 Activity Specification: An Example

Consider the *telephone-connection* task in the telecomunication application scenario given in Section 2.1.1. Assume, the initial design of the activity hierarchy for activity pattern TeleConnect, as shown in Figure 1(a). It consists of four constituent activity patterns (templates). Three simple activity patterns: A_1:ClientRegister, A_2:CreditCheck, A_3:InstallNewCircuit, and one composite activity pattern B:AllocateCircuit(ClientId, Start, End, Circuit-Id), where Start and End are Ids of end points of the circuit and ClientId represents the customer identifier. Following the activity dependencies presented in Section 2.3.2, either B or A_3 will result in a completed circuit. Thus, only one should be allowed to complete.

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

Behavioral Aggregation of Constituent Activities:
A_1: ClientRegister(*In:* ClientId:CLIENT, Start:POINT, End:POINTII)
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 Rules:
$ExeR_1$: A_1 **precede** A_2
$ExeR_2$: A_1 **precede** $[B, A_3]$

Interleaving Rules:
ILR_1: $\{A_1, A_2\}$ **precede** A_{10}

State Transition Rules:
STR_1: *abort*(B) **enable** *commit*A_3
STR_2: *commit*(B) **disable** A_3
STR_3: *commit*(B) **disable** *active*(A_3)

end Activity

Figure 5: An example specification of composite activity TeleConnect

Note that B and A_3 are non-commuting subactivities, but they are still compatible in terms of the specific semantics of this application. To allocate a circuit from existing resources, the first thing (that) activity B does is to allocate lines between the two end points and the nearest central offices (Allo-

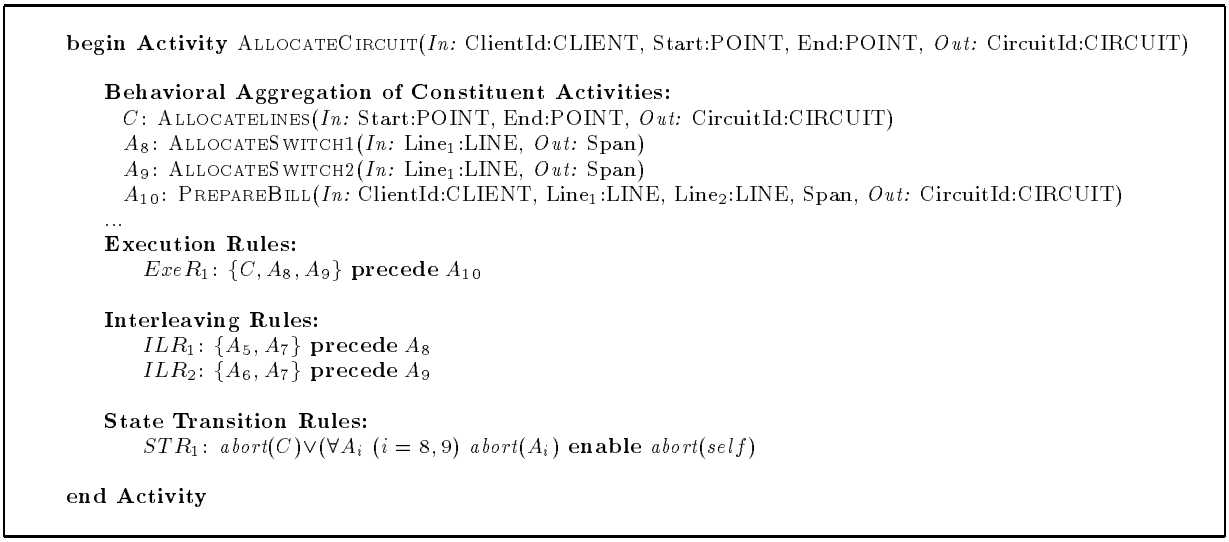

Figure 6: An example specification of composite activity AllocateCircuit

cateLine1, AllocateLine2) and trunk connection between central offices (AllocateSpan). Also to form a circuit, the end points of the allocated lines and trunks need to be connected through the switches in the central offices (AllocateSwitch1, AllocateSwitch2). Therefore, activity AllocateCircuit consists of one composite activity (C:Allocatelines) for allocating lines and trunks between two end points and three simple activities: A_8:AllocateSwitch1, A_9:AllocateSwitch2 and A_{10}:PrepareBill. The composite activity C:Allocatelines in turn consists of four simple activities A_4, A_5, A_6 and A_7. Note that several hops might be required to allocate proper trunk connection between central offices. Therefore, the activity pattern AllocateSpan can become a composite activity that consists of other activities.

A specification language can be developed based on the language we proposed earlier [21]. The example demonstrated in Figure 5 presents a sample specification of activity pattern TeleConnect. A fragment of the activity pattern AllocateCircuit is specified in Figure 6. Since the composite activity patterns TeleConnect is defined in terms of the activity specification of AllocateCircuit, the temporal execution dependencies between constituent activities of AllocateCircuit and its siblings in the activity hierarchy of TeleConnect are described as the interleaving rules in the activity specification of TeleConnect. Similarly, a fragment of the activity pattern Allocatelines is defined in Figure 5.

In addition to the syntactic parser, TAM also checks the semantic consistency of the user activity specifications, and rules out the semantic conflicts as earlier as possible. Our semantic consistency checking is primarily based on the subactivity dependency graph generated in terms of activity specifications. For example, according to the activity specifications given in Figure 7 to Figure 5, the subactivity dependency graph for the activity pattern TeleConnect is generated as shown in Figure 8. Obviously, the activity specification of TeleConnect is semantically consistent only if there is no cycle in the subactivity dependency graph. In TAM we use the subactivity dependency graphs as one of the main debugging techniques for developing the language parser.

Based on the application-specific execution dependency rules specified in AllocateCircuit and AllocateLines, we can easily obtain the compatibility of concurrent execution of subactivities of Al-

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

Behavioral Aggregation of Constituent Activities:
A_4: SELECTCENTRALOFFICES(*In:* Start:POINT, End:POINT, *Out:* Off_1:CentralOff, Off_2:CentralOff)
A_5: ALLOCATELINE1(*In:* Start:POINT, Off_1:CentralOff, *Out:* Line_1:LINE)
A_6: ALLOCATELINE2(*In:* End:POINT, Off_2:CentralOff, *Out:* Line_2:LINE)
A_7: ALLOCATESPAN(*In:* Off_1:CentralOff, Off_2:CentralOff, *Out:* Span)
...
Execution Rules:
$ExeR_1$: A_4 **precede** $\{A_5, A_6, A_7\}$

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

end Activity

Figure 7: An example specification of composite activity ALLOCATELINES

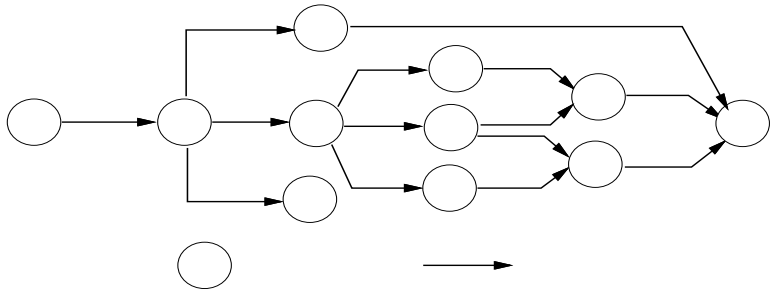

Figure 8: The subactivity dependency graph of activity pattern TELECONNECT

LOCATECIRCUIT. By combining the interleaving rules and the subactivity execution rules of TELECONNECT with the compatibility table of ALLOCATECIRCUIT, the compatibility table for concurrent execution of constituent activities of TELECONNECT can be derived, as shown in Figure 9.

The state transition rules of TELECONNECT override the state transition graph of ALLOCATECIRCUIT and INSTALLNEWCIRCUIT by adding application-specific state transition conditions into the system-default state transition graph. Figure 10 shows the two state transition graphs updated by the activity specification of TELECONNECT. The constraints within the boxes are the augmented enabling conditions for the corresponding state transitions.

3 Ensuring Correctness Criteria

The correctness criteria of an activity of pattern α describe its valid histories. An activity history is a sequence of subactivity executions, if α is a composite activity pattern (CACT), or a sequence of operation invocations through message exchanges, if α is a simple activity pattern (SACT). A valid activity history produces correct results and does not violate the consistency of objects. Since SACT activities in TAM can be viewed as traditional transactions or nested transactions, and will be mapped

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
A_1	N	N	N	N	N	N	N	N	N	N
A_2		N	Y	Y	Y	Y	Y	Y	Y	N
A_3			N	Y	Y	Y	Y	Y	Y	Y
A_4				N	N	N	N	N	N	N
A_5					N	Y	Y	N	Y	N
A_6						N	Y	Y	N	N
A_7							N	N	N	N
A_8								N	Y	N
A_9									N	N
A_{10}										N

Figure 9: The compatibility among constituent activities of TELECONNECT

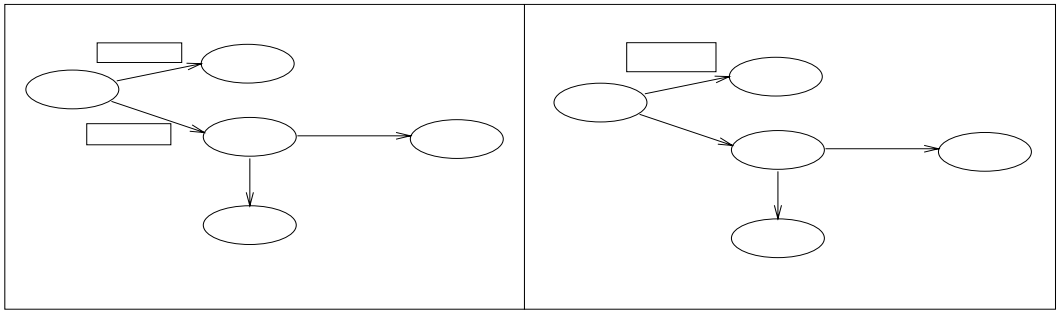

Figure 10: Augmenting the state transition graphs by means of user-defined state transition rules

to the local transactions [28, 29, 12], each running on a remote server, the correctness criteria developed in conventional transaction models [4] and nested transaction models [7] can be applied. In the sequel we concentrate on histories of composite activities.

3.1 Activity History

When a set of composite activities execute concurrently, their subactivities may be interleaved. We model such an execution by a structure called activity history. Since some of these subactivities may be executed in parallel, an activity history is defined as a *partial* order.

Definition 3 (activity history)
Let $P = \{A_1, ..., A_n\}$ be a set of activities. Each A_i $(1 \leq i \leq n)$ may again be a set of activities. A complete *activity history H over P is a two-element tuple $(H, \prec_H)$, satisfying:*

1. $H = \cup_{i=1}^{n} A_i$.
2. *The subactivities in H must not violate their activity pattern specifications and the constraints on the number of occurrences of each subactivity.*
3. $\prec_H$ *must satisfy the following properties:*

(a) $\prec_H$ *must not violate the order (denoted by* $\prec_{Exe}$*) of subactivities defined by the subactivity execution dependencies of their corresponding activity patterns. I.e.,*
$\forall A_i, A_j \in H (A_i \prec_{Exe} A_j \longrightarrow A_i \prec_H A_j)$.

(b) $\prec_H$ *must not violate the order (denoted by* $\prec_{IL}$*) of subactivities defined by the interleaving rules of their corresponding activity patterns. I.e.,*
$\forall A_i, A_j \in H \ (A_i \prec_{IL} A_j \longrightarrow A_i \prec_H A_j)$.

(c) *for any two activities* $A_i, A_j \in H$*, if* $A_i \neq A_j$ *and* **compatible**(A_i,A_j) = `false`*, then either* $A_i \prec_H A_j$ *or* $A_j \prec_H A_i$ *holds.*

(d) *for any* $A_i, A_j, A_k \in H$*, if* $A_i \prec_H A_j$ *and* $A_j \prec_H A_k$ *hold, then* $A_i \prec_H A_k$ *holds.*

4. *The subactivities in* H *must satisfy the state transition rules specified in their parent activity pattern.* □

The partial order $A \prec_H B$ can be interpreted as B depends on A, either because there exists an execution precedence rule in their parent activity pattern which requires A **precede** B (Condition(3a), or because A and B are not compatible and thus the order of A and B is important (Condition (3b) and (3c)). The transitivity of $\prec_H$ is explicitly required in the definition because neither the execution rules nor the interleaving rules can guarantee such transitivity.

An activity history $(H,\prec_H)$ is *correct* only if $\prec_H$ is a partial order over the set H of activities. It means that no activity in H indirectly depends on itself. Thus, to determine whether a history is correct w.r.t. the correctness specification, we must also consider *transitive* (*indirect*) incompatibility in addition to the direct incompatibility defined by the activity compatibility tables. The subactivity dependency graph is one of the main techniques that we use to detect the cycles (inconsistent specifications) in the users' activity pattern definitions.

A subactivity history is reflected in its parent's activity history. We encourage the activity designer to localize the correctness specifications pertaining to the subactivities of an activity. For instance, we donot want an activity to decide correctness directly for each of its children or grandchildren. Rather, we want it to decide correctness only for the cooperation of its children, including the correct execution precedence of its children activities and the correct interaction among them. With localized specification, we call an activity history *correct* (or *valid*) if it satisfies its own correctness criteria and the histories are respectively correct for each of its children activities.

To enforce a correctness specification online, we need to inform the subactivity immediately when it is executed in an invalid order. This requires that the algorithm we use to enforce the correctness criteria should be able to recognize not only the correct histories but also prefixes of correct histories.

Definition 4 (a prefix of activity history)
Let $(H,\prec_H)$ *be an activity history over the set* A *of activities* $A_1, ..., A_n$*. A partial order* $L' = (H',\prec_{H'})$ *is a* prefix *of activity history* $(H,\prec_H)$ *if and only if the following properties are verified:*

1. $H' \subseteq H$ *and for all* $A_i, A_j \in H'$, $A_i \prec_{H'} A_j$ *iff* $A_i \prec_H A_j$.

2. *for each* $A_i \in H'$*, all the predecessors of* A_i *in* $(H,\prec_H)$ *are also in* L'*. That is,*
$\forall A_k \in H$*, if* $A_k \prec_H A_i$ *holds, then* $A_k \prec_{H'} A_i$ *holds.* □

We refer to a prefix of an activity history $(H,\prec_H)$ as an *incomplete* history. We call an incomplete activity history *valid* only if it is a prefix of a *correct* activity history. In fact, this definition also presents an algorithm for checking whether an incomplete history is a valid prefix of a correct activity history.

3.2 Correctness criteria of Merged Activity Histories

When two histories $(H_1,\prec_1)$ and $(H_2,\prec_2)$ belong to the same activity pattern α and both activities have reached a breakpoint, we say that these two histories are mergable [31].

Definition 5 (a merged activity history)
Let $(H_1,\prec_1)$ and $(H_2,\prec_2)$ be two correct histories (complete or incomplete) of pattern α. We say that the history $(H_M,\prec_M)$ is a merged activity history *from $(H_1,\prec_1)$ and $(H_2,\prec_2)$, if and only if it satisfies the following properties:*

1. $H_M \subseteq H_1 \cup H_2$.
2. $\prec_M$ *is a binary relation defined over H_M, satisfying that for any two subactivities A and B in $H_1 \cup H_2$:*
   (a) *if* **compatible**(A,B) = *'true' in α, then $A,B \in H_M$ and A and B are compatible in the merged history $(H_M,\prec_M)$.*
   (b) *if* **compatible**(A,B) = *'false' in α, A and B are two different instance activities of the same pattern, then either A or B is included in the merged history but not both. I.e., $(A \in H_M \wedge B \notin H_M) \vee (B \in H_M \wedge A \notin H_M)$.*
   (c) *if* **compatible**(A,B) = *'false' in α, A and B belong to different subactivity patterns, then A and B follows an order, say A before B, and three cases need to be considered:*
      i. *If two activities A and B are from the same history, say $(H_1,\prec_1)$, i.e, $A,B \in H_1 \wedge (A \notin H_2 \vee B \notin H_2$, and $A \prec_1 B$ holds, then*
         - *B can only be included in the merged history, if A is included in the merged history. I.e., $A,B \in H_1 \wedge A \prec_1 B \wedge A \in H_M \longrightarrow B \in H_M$;*
         - *If A and B are included in the merged history and $A \prec_1 B$ holds, then $A \prec_M B$ also holds. I.e., $A,B \in H_M \wedge A \prec_1 B \longrightarrow A \prec_M B$.*
      ii. *If two activities A and B are from different histories, say $A \in H_1 \wedge A \notin H_2 \wedge B \in H_2 \wedge B \notin H_1$, then A is included in the merged history but not B.*
      iii. *If A or B are included in both histories, say $A \in H_1 \cap H_2 \vee B \in H_1 \cap H_2$, then only one A can be included in the merged history, say $A \in H_1$ is included in H_M, thus B is included in H_M only if $B \in H_1$.* □

By Condition (1) a merged history contains only subactivities that appeared in the two give histories. By Condition (2a), a merged history contains subactivities that are compatible with each other. By Condition (2b), if two activities, each from one input history, belong to the same activity pattern, then only one can be included in the merged history. By Condition (2c)(i), a merged history contains only subactivities that are allowed by the corresponding activity pattern, and each subactivity in a merged history is executed based on a state that is equivalent to the state before its execution in the orignal history. By Condition (2c)(ii) a merged history does not violate the order of subactivities determined by the execution rules and interleaving rules specified in the corresponding activity pattern. By Condition (2c)(iii), if a subactivity is contained in both histories, then only effect of one can be included in the merged history and the other has to be canceled. Also a merged history preserves the order of subactivities determined by the execution rules and interleaving rules specified in the corresponding activity pattern.

Proposition 2 *Let $(H_1,\prec_1)$ and $(H_2,\prec_2)$ be two correct histories (complete or incomplete) of pattern α. Let $(H_M,\prec_M)$ be a* merged activity history *from $(H_1,\prec_1)$ and $(H_2,\prec_2)$. Then $(H_M,\prec_M)$ is also a* correct *activity history.*

Proof
To prove that $(H_M,\prec_M)$ is a *correct* history, we need to prove that the binary relation $\prec_M$ is a partial order over H_M. I.e., $\prec_M$ is transitive and antisymmetric.

Step 1: $\prec_M$ is transitive.
For any $A, B, C \in H_M$, $A \prec_M B$ and $B \prec_M C$, to prove that $A \prec_M C$ holds, all we need is simply to prove is that A, B, C belong to the same original history, say $(H_1,\prec_1)$. Because, by Definition 3(3d), if $A \prec_1 B \wedge B \prec_1 C$ holds, then $A \prec_1 C$ holds. By Definition 5(2c)(i), if $A \prec_1 C \wedge A, C \in H_M$ holds, then $A \prec_M C$ holds.

Assume A, B, C donot belong to the same original history, say $A, B \in H_1 \wedge B, C \in H_2 \wedge A \notin H_2 \wedge C \notin H_1$, then $A \in H_1$ is included in H_M. By Definition 5(2c)(iii) only $B \in H_1$ is contained in H_M. Due to the fact that $C \notin H_1$, by Definition 5(2c)(ii) $C \notin H_M$. This contradicts with the given condition $A, B, C \in H_M$. Therefore A, B, C belong to the same input history $(H_1,\prec_1)$.

Step 2: $\prec_M$ is antisymmetric. I.e., We want to prove that if $A \prec_M B$, then $B \prec_M A$ does not hold.
By $A \prec_M B$ and Definition 5(2c)(i), A and B belong to the same original history, say $(H_1,\prec_1)$, and $A \prec_1 B$ holds. If we assume, $B \prec_M A$ holds, similarly we have $B \prec_1 A$ holds. However, according to the fact that $(H_1,\prec_1)$ is a correct history, $\prec_1$ is a partial order relation, i.e., $A \prec_1 B \wedge B \not\prec_1 A$ holds. A contradiction obtained. Hence, $\prec_M$ is antisymmetric. □

This proposition states that the correctness of a merged history is defined and guaranteed by assuming two input histories are correct. Section 5 provides a concrete example to illustrate merged activity histories and the concurrent executions of activities.

4 Dynamic Restructuring of Activities

In an open-ended cooperative environment, activities reveal two characteristics intrinsic to distributed (and 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. We introduce two types of *activity restructuring* operations, namely split-activity operations and join-activity operations, which allow users (or applications) to dynamically modify the set of concurrent activities while they are in progress. The split-activity and join-activity operations can be seen as an adaptation of the concept of split- and join-transactions proposed in [30, 19] into the TAM activity model. Our operations preserve the correctness specification of original activities. The concurrency atomicity is violated, but the application-dependent serializability (equivalence to some serial activity history) is preserved.

4.1 Split-Activity Operations

The *split-activity* 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, in which case C_i is *abort-sensitive* to C_k ($i, k \in 1, 2, 3,\ i \neq k$). When C_k aborts, if C_i is still active then C_i aborts; and if C_i is committed 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 simple activities (of SCAT patterns), a system-default option can be provided in terms of the objects accessible to each of the splitted activities, for example, a simple activity can be divided into two serializable transactions using the mechanisms defined in [30, 19]. Thus our discussion herein focuses on composite activities (of CACT patterns).

We provide three operators for "*split-by-nesting*" operations, namely **i-Split** (independent-split), **d-Split** (dependent-split), and **s-Split** (serial split); and one operator for "*split-by-unnesting*", namely **u-Split** (unnesting-split). Given a composite activity, it is possible to split a leaf node or an internal node (incl. a root node). The splitted nodes could be mutually *independent* activities, or a set of activities of which some are dependent of others (*partially-dependent*), or *serial*. Figure 11 captures the effects of splitting a leaf node activity into a set of independent, partially-dependent, or serial activities. Note that **d-Split**(C, {<C_1, C_2>,<C_3, C_2>}) is an illegal expression of partially-dependent activity-split operation, because in any activity hierarchy there exists no such subactivities C_1, C_2, C_3 that $C_1 \rightsquigarrow C_2 \wedge C_3 \rightsquigarrow C_2$ holds. Besides, to preserve the user-defined activity dependencies, the operation **u-Split** should only apply to internal node activities that are not root activities. Figure 12

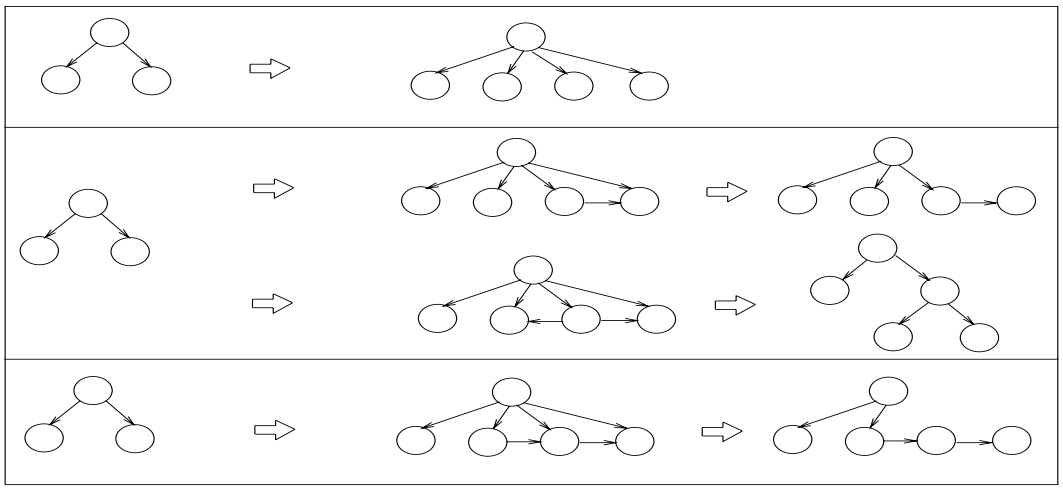

Figure 11: Splitting a leaf node activity

presents the effects of splitting an internal node activity into a set of subactivities. In these figures a solid arrow between two activities denotes an *abort-sensitive* dependency (recall Section 2.3.4). When a node C splits into two or more subactivities, say C_1, C_2, and C_3, the *abort-sensitive* dependencies between activity C and its parent activity A are assumed to hold between C_1, C_2, C_3 and A. So do the application-dependent execution dependencies and interleaving rules between C and its siblings as well as between C and children of its siblings. For example, if the activity execution rule "B **precede** C"

holds before the split operation, then we may replace "B **precede** C" using the rule

- "B **precede** $\{C_1, C_2, C_3\}$" after the operation **i-split**(C,$\{C_1, C_2, C_3\}$) is performed.
- "B **precede** $\{C_1, C_2\}$" after the operation **d-split**(C,$\{C_1$, <C_2, C_3>$\}$) is performed.
- "B **precede** C_1" after the operation **s-split**(C,$\{$<C_1, C_2, C_3>$\}$) is performed.

Similar treatment are applicable to user-defined activity interleaving rules and activity state transition rules.

A node splitting may result in a subactivity that has *abort-sensitive* dependencies on two or more other subactivities (see Figure 11 and Figure 12). Such inconsistencies may be reduced (resolved) by applying the following consistency-preserving rewriting rule (see Figure 11), because this consistency-preserving rewriting rule may help to eliminate redundant dependencies and thus simplify the structure of a complex activity.

$$A \rightsquigarrow B \wedge B \rightsquigarrow C \wedge A \rightsquigarrow C \iff A \rightsquigarrow B \wedge B \rightsquigarrow C,$$

where $A \rightsquigarrow B$ denote that A is *abort-sensitive* to B, i.e., if B aborts

then A aborts if A is active, and A is compensated if A has committed.

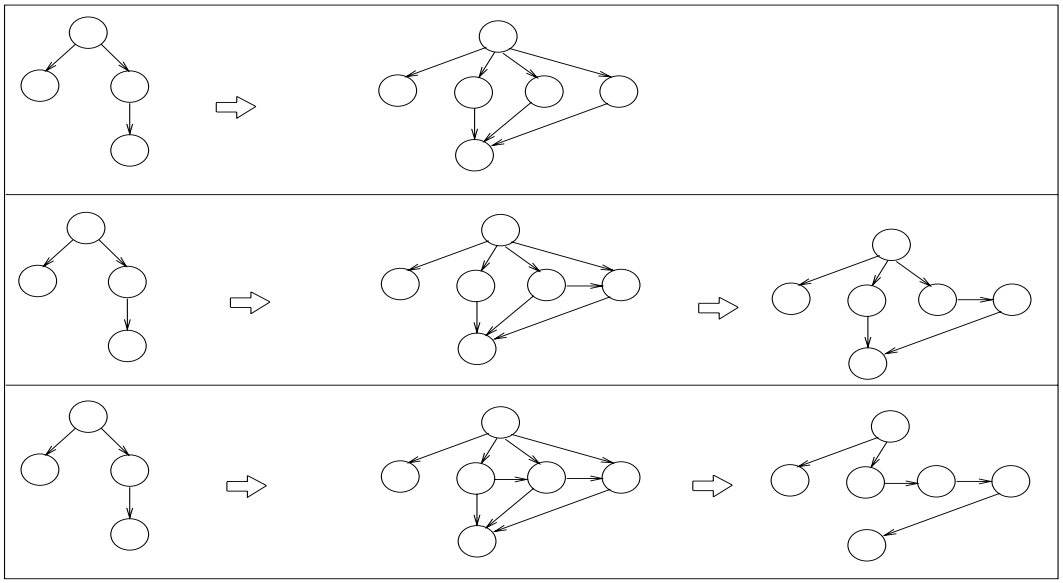

Figure 12: Splitting an internal node activity

After applying the rewriting rule, we examine the remaining dependencies for each type of split-activity to see if the resulting structure preserves the semantics of the original activity hierarchy. We conclude that, the properties of the original activity, such as abort-sensitive dependencies, activity execution and interleaving dependencies, are preserved in all the cases of leaf node splitting. Unfortunately, for internal node splitting, only *serial-split* operation preserves the *abort-sensitive* dependencies. Thanks to the dynamic nature of the split-activity operations, an internal node may become a leaf node at any

point after all its *active* subactivities have terminated and before activating any new subactivities. For example, the independent-splitting of activity C in Figure 12, namely **i-split**(C, $\{C_1, C_2, C_3\}$), may proceed when subactivity D terminates. Thus subactivities C_1, C_2 and C_3 may commit independently. When C_1 commits earlier than C_2 and C_3, the objects that C_1 delegated to A are potentially accessible to B. This effectively improves the cooperation between C and its orignal siblings such as B, since B can access the objects modified by C_1 while C_2 and C_3 are still executing.

Interesting to note is that the nested-split transaction model discussed in ACTA [7] only considers independent and serial splitting of transactions. Moreover only independent splitting of leaf node transactions is allowed in terms of consistency. In the TAM activity model, we consider not only independent and serial splitting of activities but also partially-dependent splitting of activities which happens mostly frequently in practice. Furthermore, all cases of leaf-node splitting preserve the dependencies of the original activity. The serial splitting of internal node activities is also permitted in terms of consistency. We believe that the combination of TAM and the dynamic split-activity operations provides a set of useful facilities for organizing open-ended cooperative activities in an open distributed environment where cooperation is emphasized and promoted.

4.2 Join Activities

The inverse operation of split-activity, called *join-activity*, can combine results 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 supported. The operation "*join-by-nesting*" (**g-join**) groups two or more activities by creating a new activity as their parent activity. The operation "*join-by-unnesting*" (**m-join** merges two or more activities into one single activity. Figure 13 presents the join-activity operation for grouping or merging activities at same abstraction level (either leaf or internal node). For grouping or merging activities of different abstraction levels, a combination of split and join operations can be used. This restriction allows the automated enforcement of correctness specification in the event of dynamic restructuring. To ensure the correctness criteria of join-activity operations,

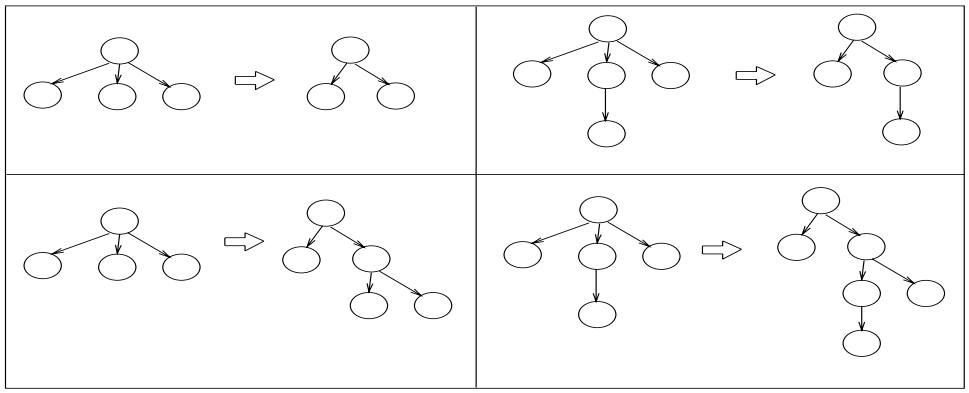

Figure 13: Join-activity operations

the merged or grouped activities must preserve all application-specific activity dependencies, such as

subactivity execution rules, subactivity interleaving rules and state transition rules. The implementation algorithm for join-activity operations can easily be developed in terms of Definition 5 and Proposition 2.

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 transactional activity model for organizing open-ended cooperative activities in distributed and multi-user design and computing environment.

5 Concurrent Executions of Activities: Example Continues

In this section we illustrate by example how to produce a *correct* merged history from two concurrently executed histories, and how the concepts discussed in this paper, such as the subactivity execution rules, the subactivity interleaving rules, and the state transition rules are used as the correctness criteria for execution of an activity of pattern TeleConnect, and how to apply various split- and join-activity operations to obtain added concurrency and cooperation. We assume that instance activities of pattern α are uniquely identified [20]. Also each user activity history has a private workspace. A user may delegates his/her work from the current workspace to the workspace of another user activity history. Delegation means that the user is passing work that is possibly incomplete to another user.

Consider the following scenario in which three different users participate in the execution of an activity of pattern TeleConnect described in Figure 6. All the three users cooperate, in accomplishing the task of telephone installation, by performing various subactivities related to the task.

1. $User_1$ starts activity `a` of pattern TeleConnect upon receiving a service request. She first executes the subactivity `a1` of type A_1:ClientRegister, and then the subactivity `a4` of type A_4:SelectCentralOffices. After completion of these subactivities, her user activity history is:

   $User_1$: `<a1, a4>`.

   Before $User_1$ delegates her work to $User_2$ and $User_3$, the operation **u-Split**(C, $\{A_4, A_5, A_6, A_7\}$) is invoked by $User_1$ to split the activity `c` of pattern AllocateLines through unnesting, in order to allow releasing of early committed resources to be accessible to other subactivities that are concurrently executed, such as A_8 and A_9. By splitting the activity `c` of pattern C, whenever A_5 and A_7 both commit, A_8 can be executed without waiting for A_6 to terminate, since the results of A_5 and A_7 are accessible to A_8 as soon as A_5 and A_7 commit. The Πoperation **delegate**($User_1$,$\{User_2, User_3\}$) enables $User_1$ and $User_2$ to delegate their work to $User_1$. The activity history of $User_1$ is then merged with the activity history of $User_2$ and with the activity history of $User_3$ respectively.

2. $User_2$ accepts the work being delegated by $User_1$ and merges the work of $User_1$ into his history, since both histories belong to the same pattern TeleConnect and thus are mergable. $User_2$ then executes `a6` of A_6:AllocateLine2, `a7` of A_7:AllocateSpan, and `a9` of A_9:AllocateSwitch2. $User_2$'s history looks like this:

   $User_2$: `<a1, a4, a6, a7, a9>`.

3. $User_3$ starts his work based on the work delegated by $User_1$. Then he executes the subactivity `a5` of A_5:AllocateLine1, and in the meantime, he executes concurrently the subac-

tivity a6' of A_6:AllocateLine2, the subactivity a7' of A_7:AllocateSpan, and then a2 of A_2:CreditCheck, and a8 of A_8:AllocateSwitch1. The history of $User_3$ is shown below:

$User_3$: <a1, a4, a5, a7', a2, a8>.

Upon completing these subactivities, $User_3$ decides to delegate his work to $User_2$.

4. $User_2$ accepts the work being delegated and merges the history of $User_3$ with his own work. By Definition 5(2b), the $User_2$ has to choose to accept either a7 or a7' but not both. Suppose $User_2$ chooses to accept a7', by Definition 5(2c) a8 is accepted but a9 has to be redone. This is because (i)a9 depends on a6 and a7, (ii)a8 depends on a5 and a7'. After the delegation, the history of $User_2$ looks like this:

   $User_2$: <a1, a4, a5, a6, a7', a2, a8>.

5. After re-execution of subactivity a9 of A_9:AllocateSwitch2, $User_2$ delegates his work to $User_1$ and exits.

   $User_2$: <a1, a4, a5, a6, a7', a2, a8, a9'>.

6. $User_1$ resumes her work based on the merge of $User_2$'s history with her own. By Definition 5(2a)(2b), the merge is straightforward. $User_1$ then performs activity a10 of pattern PrepareBill, and completes the task of TeleConnect. The final history of $User_1$ will be:

   $User_1$: <a1, a4, a5, a6, a7', a2, a8, a9', a10>.

Observe that the final result of the concurrent executions of subactivities of the activity a of type TeleConnect is equivalent to the execution of activity a by a single imaginary user in any order, that conforms to the dependency graph of Figure 8, without any interferences or interleavings. No activity dependency rules specified in TeleConnect of Figure 6 are violated.

6 Conclusion

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] A. Buchmann, M. Ozsu, M. Hornik, D. Georgakopoulos, and F. Manola. A transaction model for active distributed object system. In *Elmagarmid [?]*, pages 123–158, Chapter 5, 1992.

[7] P. Chrysanthis and K. Ramamritham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 194–203, 1990.

[8] P. Chrysanthis and K. Ramamritham. Acta: The saga continues. In *Elmagarmid [?]*, pages 349–397, 1992.

[9] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

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

[11] 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.

[12] A. Deacon, H. Schek, and G. Weikum. Semantic-based multilevel transaction management in federated systems. In *Proceedings of International Conference on Data Engineering*, pages 452–461, 1994.

[13] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, 1990.

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

[15] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM SIGMOD Int. Conference on Management of Data*, pages 462–473, 1987.

[16] 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.

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

[18] J. Gray. The transaction concept: Virtues and limitations. In *Proceeding of the 7th International Conference on Very Large Databases*, pages 144–154. Morgan Kauffman, 1981.

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

[20] L. Liu. An formal approach to structure, algebra, and communication of complex objects. Technical report, Ph.D dissertation, Tilburg University, ISBN 90-9005694-7, 1992.

[21] 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*, Vancouver, Canada, 1992. Morgan Kauffman.

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

[23] C. Mohan. *Advanced Transaction Models - Survey and Critique*. Tutorial presented at the ACM SIGMOD international conference, 1994.

[24] 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.

[25] E. Moss. *Nested Transactions*. Cambridge, Mass., MIT Press, 1985.

[26] M. Nodine, S. Ramaswamy, and S. Zdonik. A cooperative transaction model for design databases. In *Elmagarmid [?]*, pages 53–85, Chapter 3, 1992.

[27] 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.

[28] M. Ozsu, U. Dayal, and P. Valduriez, editors. *Distributed Object Management*, Edmonton, Canada, August 1992. Morgan Kaufmann.

[29] C. Pu and S. Chen. Implementation of a prototype superdatabase. In *Proceedings of the Workshop on Experimental Distributed Systems*, Huntsville, Alabama, October 1990.

[30] 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.

[31] 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.

[32] J. Wasch and A. Reuter. The contract model. In *Elmagarmid [?]*, pages 219–264, Chapter 7, 1992.