A Transactional Activity Model for Organizing Open-ended Cooperative Activities

Ling Liu and Calton Pu
Dept. of Computer Science and Engineering
Oregon Graduate Institut of Science and Technology
P.O.Box 91000 Portland, Oregon 97291-1000 USA
{lingliu,calton}@cse.ogi.edu

Abstract

We propose a transactional activity model (TAM) for specification and management of open-ended cooperative activities. In contrast to each of the extended transaction models which focus more on a formal template for data processing patterns in an application, the activity model contains a set of formal templates, namely activity patterns, for capturing patterns of object interactions in an application. Using TAM, users may form new activity patterns by combining components of different existing patterns using activity pattern refinement and activity pattern composition mechanisms that include split and join of activities. In TAM, activity histories are used to study the correctness of activity execution and activity restructuring through refinement, composition, split, or join.

1 Introduction

Open-ended activities are characterized by long duration, flexible cooperation and dynamic development [20]. The need to support open-ended activities emerges from the increasing demands of advanced and complex applications, such as CAD/CAM, automated office workflows information systems, and software development environments. Activities in these 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.

In this paper, we develop a transactional activity model (TAM) for specification and management of open-ended cooperative activities. TAM is a refinement of the activity model proposed in [13] by adding transactional properties and by incorporating dynamic restructuring operations [20] into the activity execution. Our results in this paper are a significant advance over each one of the component technology we used. For example, by adding the semantics of compensation and exception handling of activities to the original activity model, we are able to add the support for failure recovery and correct cooperation among concurrent users. Another example of features supported by TAM is the *activity patterns*, which can be defined statically and modified dynamically through the invocation of system-supplied activity restructuring operations in the progress of activities. In contrast to the transaction models which focus more on formal templates for data processing patterns in an application, the activity model consists of formal templates, namely activity patterns, that describe patterns of object interactions in an application.

To address the needs of open-ended and cooperative tasks, TAM provides modeling primitives that allow activity model designers to create implementation-independent specification of complex transactional activities. We provide a number of facilities for 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 [6, 21]. TAM also supports the behavioral refinement and behavioral composition of activity patterns to increase the extensibility 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 [4, 12]. The reasoning capability has also been tested by using the TAM to study the new properties of activity split in the context of TAM.

We discuss related work in Section 2. Section 3 introduces the TAM activity model, including the formal definition of simple and composite activity patterns, and the modeling primitives for specification of user-defined activity execution dependencies, interleaving dependencies, and state transition dependencies. Section 4 discusses issues related to ensuring correctness criteria. We outline the implementation framework in Section 5 and conclude the presentation in Section 6.

2 Related Work

The development of TAM can be seen as an extension of the activity model proposed in [13] by adding transactional properties and incorporating dynamic restructuring operations [20] into the activity execution. Our early proposal [13] focused more on the scheduling of message exchanges within an execution of simple (flat) activities. In TAM, transactional properties are introduced for the hierarchical organization of activities. We also define the semantics of compensation and exception handling of activities, thus supporting failure recovery and correct cooperation among concurrent users. 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 the promotion of enhanced intra-activity parallelism by allowing an activity to be modified dynamically through the invocation of system-supplied activity restructuring operations.

In addition to our early proposal [13], other activity models have been proposed [8, 21] to support declarative specification of control flows within activities. Features of long running activities [8, 7] 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 [21] achieves cooperation by controlled data exchange of the content of workspaces and controlled sharing of a common database among users. However, these models provide no or less sufficient high-level activity abstraction mechanisms for behavioral refinement and behavioral composition of activity patterns, including split and join of activities. Hence, issues such as correctness specification and ensurance criteria for concurrent execution of activities and dynamic restructuring of activities have been left open in most of the previous activity models.

On the other hand, a number of *extended transaction models* (ETMs) and workflow systems [10, 11, 16, 17, 22, 20, 9] have been proposed to support diversified new application requirements. For example, Sagas [11] extend the traditional transaction model by including an automated compensation capability within sagas. Cooperative Transactions [2], Split Transactions [20] and Transaction Groups [17] were proposed for capturing the interactions among transactions required in advanced applications. Many of these ETMs were targeted at a particular domain of applications, offering adequate correctness only for this domain [6]. Other ETMs capture only a subset of the spectrum of interactions found in any complex information systems [15]. Cooperation among sibling transactions is supported by Cooperative Transactions [2, 20, 17], although 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. Therefore, it is usually 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 [6], and allow 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.

3 The Transactional Activity Model

3.1 Activity Patterns

Activities are executions of activity patterns. An activity pattern describes the communication protocol of a group of cooperating objects in accomplishing a task [13]. Given a universe of discourse (UoD), an activity model consists of a nonempty set of activity patterns, describing the intended communication behaviors of cooperating objects in the UoD. The transactional activity model provides a number of modeling primitives to allow activity model designers to create implementation-independent specification of complex transactional activities in terms of activity dependencies.

In TAM, each concrete activity is described by a single activity pattern and can be seen as an instantiation of the activity pattern. To facilitate the abstract specification of the hierarchical organization of *open-ended* activities, TAM distinguishes two types of activity patterns: *simple activity patterns* and *composite activity patterns*. A simple activity of patterns is a program that issues a stream of messages to access the underlying database. A composite activity pattern is a tree of subactivities, cooperating to accomplish a specific task. The subtrees of a complex activity are either composite activities or simple activities. Activities at leaf nodes are simple activities. The distance from the root to the leaf nodes can be different from different parts of the tree. The activity at the root of the tree is called *root activity*, the others are called *subactivities*. An activity's predecessor in the tree is called a *parent*; a subactivity at the next lower level is called a *child*. 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*. 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.

Activity patterns that are defined by combining components of different existing activity patterns using activity pattern refinement (behavioral specialization) and activity pattern composition (behavioral aggregation) mechanisms are called *complex activity patterns*. Two types of additional information are required for defining a composite activity pattern:

- The set of activity execution dependencies, including both the temporal execution precedence and the interleaving dependencies.
- The set of application-specific activity state-transition dependencies.

In this paper we discuss activity patterns primarily in terms of the set of constituent activities, the set of activity execution dependencies and the set of activity state dependencies, because our focus is on the transactional properties of TAM, rather than the complete activity specification language of TAM.

3.2 A Workflow Example

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, 12]. Suppose the activity A:TELECONNECT consists of four subactivities A_1:CLIENTREGISTER, A_2:CREDITCHECK, B:ALLOCATECIRCUIT, and A_3:INSTALLNEWCIRCUIT.

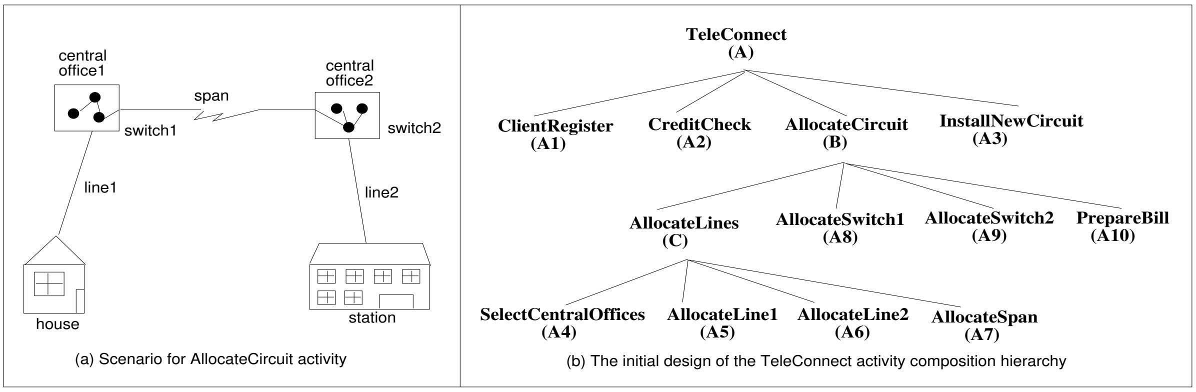

Figure 1: The telephone line connection application

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(a) illustrate the scenario for AllocateCircuit and Figure 1(b) presents a possible initial task breakdown of the activity TeleConnect. A_i $(i = 1, ..., 10)$ are elementary activity patterns, and A:TeleConnect, B:AllocateCircuit and C:SelectConnections are composite activities.

Figure 2 and Figure 3 present the TAM activity specifications for this example. First, 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. Second, users may add additional activity dependencies that are domain-dependent to the activity specification. TAM classifies application-specific interaction dependencies into three categories: activity execution dependencies, activity interleaving dependencies, and activity state transition dependencies.

Another feature of TAM is that the specification scope of the set of activity dependencies associated with each activity pattern is regulated to encourage incremental specifications of hierarchically complex activities. For instance, to define an activity pattern T, the activity model designer needs to focus on (specify) only the activity execution dependencies between activities that are children of a T activity, and restrict the activity interleaving dependencies specified in T to be only the interaction dependencies between (immediate) subactivities of different child activities of T or between a T's child activity and (immediate) subactivities of its siblings. Furthermore, although an activity designer may not override the termination-sensitive state transition dependencies (recall Section 3.3.1) between a T activity and its children activities, but they can add additional application-specific state transition dependencies, such as commit of a child activity is critical to the commit of its parent activity (see Figure 3).

To guarantee the consistency of the activity dependency specification, the TAM activity specification parser will generate an activity dependency graph, as shown in Figure 4, from the activity specifications given in Figure 2 and Figure 3. If there is a cycle, the TAM parser will report the inconsistency of the given set of activity specifications.

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

    Behavioral Aggregation of component Activities:
      A1: ClientRegister(In: ClientId:CLIENT, Start:POINT, End:POINT)
      A2: CreditCheck(In: ClientId:CLIENT, Start:POINT, End:POINT, Out: creditStatus:Boolean)
      A3: 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:
        ExeR1: A1 precede B, A3

    Interleaving Dependencies:
        ILR1: A2 precede A10

    State Transition Dependencies:
        STR1: abort(B) precede commit(A3)
        STR2: commit(B) disable A3
        STR3: commit(B) disable begin(A3)

end Activity
```

Figure 2: An example specification of composite activity TeleConnect

3.3 Activity Dependencies

3.3.1 State Transition Dependencies

An 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.

Similar to the saga transactions [11], the *commit* of a subactivity in TAM is independent of the *commit* of its parent activity. If an activity aborts, then all its children that are in *begin* state are aborted; and its *commit*ted children, however, are compensated for. We call this property *termination-sensitive* dependency between an activity A_C and its parent activity A_P, denoted by $A_P \leadsto A_C$. This termination-sensitive dependency, inherent by an activity hierarchy, prohibits a child activity instance from having more than one parent, ensuring the hierarchically nested structure of active activities. When the abort of all active subactivities of an activity is completed, the compensation for committed subactivities is performed by executing the corresponding compensations in an order that is the reverse of the original order.

3.3.2 Basic Specification Constructs

In TAM application-specific activity dependencies are user-defined and are associated with composite activity patterns. They capture complex interaction dependencies between activities that have no hierarchical composition relationship with each other. Examples include constraints on the occurrence of a subactivity execution and the temporal precedence of execution of subactivities at the same abstraction level and across different levels in the activity composition hierarchy. 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

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

end Activity

Figure 3: An example specification of activities AllocateCircuit and AllocateLines

for specification convenience since two TAM activities are compatible when there is no execution precedence dependency between them. For reference convenience, the semantics of each construct is formally described in Figure 5. Note that the semantics given in Figure 5 shows that **enable** and **disable** are not derivable from each other.

Consider the telecommunication workflow example. The dependency Exe_1 says that B, A_3 cannot begin (be *active*) before A_1 commits. The dependency STR_1 implies that A_3 can only commit after B *abort*s. That is, whether A_3 eventually commits or not is an orthogonal matter.

4 Ensuring Correctness

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 to the local transactions [18, 19, 9], each running on a remote server, the correctness criteria developed for conventional and extended transaction models [5, 6] can be applied. In this section, we only define the concept of histories of composite activities, and adapt the notation from [5] for histories of simple activities.

4.1 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 1 (activity history)
Let $P = \{A_1, ..., A_n\}$ be a set of activities. Each A_i $(1 \le i \le n)$ may again be a set of activities. A complete *activity history H over P is a two-element tuple $(H, \prec_H)$, satisfying:*

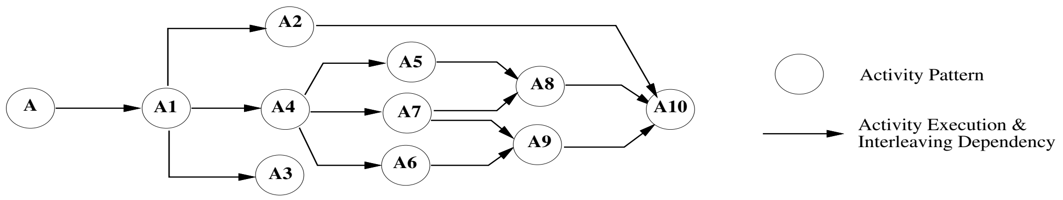

Figure 4: The activity dependency graph derived from activity pattern TELECONNECT

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

Figure 5: Constructs for activity dependency specification in TAM

1. $H = \cup_{i=1}^{n} A_i$.
2. *The subactivities in H must not violate the constraints on the number of occurrences of each subactivity specified in their activity patterns.*
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 *valid* with respect to the activity specification 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 valid 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 do not 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 *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 on-line, 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 2 (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 $A_k \in H' \wedge A_k \prec_{H'} A_i$ holds.* □

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

4.2 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* [21].

Definition 3 (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 $H_1 \cup H_2$:*

1. *if* **compatible**(A, B) = *'true' in α, then $A, B \in H_M$ and A and B are compatible in the merged history $(H_M, \prec_M)$.*
2. *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., $(A \in H_M \wedge B \notin H_M) \vee (B \in H_M \wedge A \notin H_M)$.*
3. *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:*
   - *(a) Consider two activities A and B from the same history, either $(H_1, \prec_1)$ or $(H_2, \prec_2)$. Without loss of generality, we assume $(H_1, \prec_1)$, i.e., $A, B \in H_1$ 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., $A, B \in H_M \wedge A \prec_1 B \longrightarrow A \prec_M B$.*
   - *(b) 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.*
   - *(c) 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 if $B \in H_1$.* □

By Condition (1) a merged history contains only subactivities that appeared in the two given 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 original 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 1 *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 1(3d), if $A \prec_1 B \wedge B \prec_1 C$ holds, then $A \prec_1 C$ holds. By Definition 3(2c)(i), if $A \prec_1 C \wedge A, C \in H_M$ holds, then $A \prec_M C$ holds.

Assume A, B, C do not 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 3(2c)(iii) only $B \in H_1$ is contained in H_M. Due to the fact that $C \notin H_1$, by Definition 3(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 3(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 valid. Due to the space restriction, we here omit examples that illustrate the concept of correctness of merged activity histories. Readers may refer to [14] for detail.

5 Implementation Issues

The implementation of TAM is a subject of our ongoing research. In this section, we outline some of the major issues faced by implementors and some potential solutions.

A simple activity model contains elements similar to those found in a workflow system such as Exotica [16], including reliable/recoverable queues and a state machine interpreter. However, full recovery of activities during execution is non-trivial and that is one of places where transaction software can help.

Although not a direct implementation of TAM, Barga and Pu [4] have developed a method, called the Reflective Transaction Framework to implement extended transaction models by carefully extending on-line transaction processing (TP) monitors. Concretely, they have implemented several such models on Transarc Encina, a commercial TP monitor. The models implemented [4] include split and join transactions and cooperative groups, intended for applications that require cooperation. These models are precisely the ETMs of major interest to TAM.

The Reflective Transaction Framework [3] also includes a systematic way to implement semantics-based concurrency control methods in on-line TP monitors. The basic idea is to explicitly codify the conflicts defined to be allowable by each semantics-based concurrency control in a compatibility table. The extended TP monitor then consults the appropriate compatibility tables during execution, when a conflict arises. If the conflict is found to be allowable, then the TP system lets the execution proceed. Otherwise, the conflicting transaction are stopped. In TAM, the compatibility table for activities is derived by the system (see Figure 2 and Figure 3). Thus the system can incorporate these activity-specific compatibility tables and consult them when conflicts arise. We are currently working towards building a TAM adapter based on the micro-protocols developed by [4, 3, 23] on top of the Transarc Encina.

Finally, we would like to mention that TAM development is a continuation and a careful combination of our earlier work on activity model [13] and split-transactions [20], and inspired by the recent work in [21]. Comparing with the work in [21], TAM provides a number of distinct features. For example, TAM users may form new activity patterns by combining components of different existing patterns using activity pattern refinement and activity pattern composition mechanisms that include split and join of activities footnoteDue to the space limitation, we omit the discussion on TAM activity restructuring operations. Interested readers may refer to [14] for detail.. The concept of activity history and merged activity history are formally defined. Our formal definition of a merged history and the proof that a merged history is valid if the two input activity histories are valid also present a practical algorithm to construct a valid history that merges two existing histories. TAM not only supports various cooperation needs required at different stages of a task execution, but also guarantees the correctness of concurrent execution of user activities and the reliability of dynamic split and join of user activities with respect to the given activity specifications.

6 Concluding Remarks

We have proposed the transactional activity model TAM as an approach to transactional support for open-ended and cooperative applications. In TAM, *open-ended*ness is achieved by supporting a number of dynamic restructuring operations for activity-splits and activity-joins. Cooperation is achieved by regulated interleaving among subactivities and controlled data exchange among concurrent execution of user activities. Application-specific activity dependencies are served as the correctness specification for regulation of object interaction. One of the key concepts that TAM promotes is the separation of activity specifications from their concrete implementations at individual servers, which allow reasoning about transactional properties of open-ended activities independently from their local transaction implementations. TAM also provides a number of mechanisms to facilitate the specification and reasoning of application-specific activity dependencies. Our model ensures that a joint execution of a cooperative activity is equivalent to the one that could have been carried out by a single user (a single client or a single server). The concept of complete activity history, a valid prefix of activity history, and a merged activity history are formally introduced to study the transactional properties of dynamically joint activities. Our activity merge algorithm guarantees that a merged history from two valid histories is valid with respect to the two input histories. All these features make TAM a valuable tool for applicable designers.

Our research on TAM continues along several lines. On the formal side, we are exploring a more concise description of TAM including exploring the potential of the ACTA framework. On the implementation side, we are extending the Reflective Transaction Framework to design a practical implementation of TAM using production TP software. On the application side, we are looking into real world applications that can benefit from the adaptive features of TAM and guide its further development.

Acknowledgment

The first author was partially supported by NSERC grant OGP-0172859 and STR-0181014 and a grant from APRA. The second author is partially supported by NSF grant IRI-9510112, a APRA grant, and grants from Intel and Hewlett-Packard.

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. *A Reflective Framework for the Implementation of Extended Transactions*. PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, Expected, 1997.

[4] 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.

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

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] L. Liu and C. Pu. Methodical restructuring of complex workflow activities. In *Proceedings of the 14th International Conference on Data Engineering (ICDE'98)*, Orlando, Florida, USA, February 1998.

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] J. Wasch and A. Reuter. The contract model. In *Elmagarmid [10]*, pages 219–264, Chapter 7, 1992.

[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.