Methodical Restructuring of Complex Workflow Activities *

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

We describe a family of activity-split and activity-join operations with a notion of validity. The key idea of introducing the set of activity-split and activity-join operations is to allow users to restructure ongoing activities in anticipation of uncertainty so that any significant performance loss due to unexpected unavailablity or delay of shared resources can be avoided or reduced through release of early committed resources or transferring ownership of uncommitted resources. To guarantee the correctness of new activities generated by activity-split or activity- join operations, we define the notion of validity of activity restructuring operations and identify the cases where the correctness is ensured and the cases where activity-split or activity-join are illegal due to the inconsistency incurred.

1 Introduction

Workflow systems have evolved to support activities of hierarchically nested structure and sophisticated interaction dependencies. Such complex activities are vulnerable to uncertain duration and unpredictable development. To address these challenges, we introduce a set of activity-split and activity-join operations for dynamic restructuring of complex workflow activities in the context of TAM. We refer to the set of activity-split meta operations as *ASplit* and the set of activity-join meta operations as *AJoin*. *ASplit* operations are specialized into: (1) a *serial split* by a **precede** dependency, which requires a serial execution, (2) a *parallel split* by a **compatible** dependency, and (3) an *unnesting split* otherwise. Similarly, *AJoin* operations are specialized into *join-by-group* and *join-by-merge*. Comparing with the early research on split-transactions [18] and Cooperative Transactions [1, 17], these activity restructuring meta operations are novel in at least two aspects. First, a formal notion of the validity of activity restructuring operations is introduced to guarantee the correctness of new activities generated by activity-split or activity-join operations. Second, the techniques of specialization classes are used to define user-driven activity splits and joins. By specializing the activity restructuring meta-operations for actual execution, an efficient implementation of various split and join operations can be explored.

The rest of the paper proceeds as follows. We first describe the reference activity model TAM in Section 2. Then we introduce the family of activity restructuring meta-operators and study the validity and benefits of activity-split and activity-join operators in Section 3. We discuss implementation issues and related work in Section 4.

2 The Reference Activity Model - TAM

2.1 Activity Patterns and Activity Hierarchy

Activities in TAM 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 seen as an instantiation of its activity pattern. Informally, an activity pattern consists of objects, messages, message exchange constraints, preconditions, postconditions, and trigger conditions [11].

TAM distinguishes two types of activity patterns: *elementary* activity patterns and *composite* activity patterns. An elementary activity pattern is a program that consists of transaction operations such as *Begin*, a sequence of *Read* and *Write* operations, and *Commit* or *Abort*. We refer to an execution of an elementary activity pattern as an elementary activity. A *composite* activity pattern consists of a tree of composite or elementary activity patterns and a set of user-defined activity dependencies. The tree organization of an activity pattern α is called *activity composition hierarchy* (or activity hierarchy) of α. We refer to the activity patterns that are components of a composite activity pattern as *constituent* activity patterns. The set of activity dependencies associated with a composite activity pattern can be seen as the cooperation contracts among activities that collaborate in accomplishing a complex workflow task. We use *complex activity* or *composite activity* to refer to an execution of a composite activity pattern and *subactivity* to refer to an execution of a constituent activity pattern within a composite activity. The activity at the root of the tree is called *root activity*, the others are subactivities. An activity's predecessor in the tree is called a *parent*; a

*This research is partially supported by ARPA grant MDA972-97-1-0016, NSF grant IRI-9510112, and a grant from Intel

subactivity at the next lower level is called a *child*. TAM allows arbitrary nesting of activities since it is generally not possible to determine a priori the maximum nesting an application task may need. In the rest of the paper the term *activity* (or *subactivity*) and the term *activity pattern* (or *constituent activity pattern*) are used interchangeably when no conflict incurs.

2.2 Activity 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 [8], 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 compensation transactions in an order that is the reverse of the original order.

Definition 1 (TAM activity)
Let α denote an activity pattern and Σ denote a set of activity patterns. Let $AD(\alpha)$ denote a set of activity dependencies specified in α, $children(\alpha)$ denote the set of child activity patterns of α, and $Pattern(T)$ denote the activity pattern of activity T. An activity T is said to be a TAM activity if and only if it satisfies the following conditions:

- $\exists \alpha \in \Sigma$, $Pattern(T) = \alpha$.
- $\forall P \in AD(\alpha)$, $P(T) = true$.
- $\forall S \in children(T)$, $T \leadsto S$ *and* S *is a TAM activity.*

2.3 Activity Visibility Dependencies

The visibility of an activity refers to the ability of an activity to see the results of other activities while it is executing. The rule governing the visibility of object versions between a composite activity A_P and its child subactivity A_C is described as follows:

- **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 TAM, the visibility of an activity is defined in terms of the object access set of its parent activity, which forms a visibility dependency between the given activity and its parent activity. In TAM, we use the multiple object version scheme to implement the notion of visibility in the presence of concurrent execution of activities [14].

2.4 Specification of Activity Dependencies

In TAM application-specific activity dependencies are user-defined. 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 temporal precedence of the execution of subactivities at the same abstraction level or across different levels in the activity composition hierarchy [12].

TAM provides four constructs for specification of user-defined activity dependencies. They are **precede, enable, disable**, and **compatible**. The construct **precede** is designed to capture the temporary precedence dependencies and the existence dependencies between two activities. For example, "A **precede** B" specifies a *begin-on-commit* execution dependency between the two activities: "B cannot begin before A commits". The constructs **enable** and **disable** are utilized to specify the enabling and disabling dependencies between activities. One of the critical differences between the construct **enable** or **disable** and the construct **precede** is that **enable** or **disable** specifies a triggering condition and an action being triggered, whereas **precede** only specifies an execution precedence dependency as a precondition that needs to be verified before an action can be activated, and it is not an enabling condition that, once satisfied, triggers the action. The construct **compatible** declares the compatibility of activities A_1 and A_2. It is provided solely for specification convenience since two TAM activities are compatible when there is no execution precedence dependency between them. For reference convenience, the semantics of each construct is formally described in Figure 1.

3 Dynamic Restructuring of Activities

Activities in advanced workflow applications are often long-running activities that exhibit one of the following characteristics: (1) some data may be modified early by initial subactivities and then remain unchanged until commit; and (2) some (or all) of the later subactivities may become independent of the previous subactivities. In either of the cases, it is desirable for the results of earlier

Construct	Usage	Synopsis
precede	A_1 **precede** A_2 condition(A_1) **precede** A_2 condition(A_1) **precede** condition(A_2)	A_2 can *begin* if A_1 *commits* A_2 can *begin* if *condition*(A_1)='true' holds. If *condition*(A_1)='true' then condition(A_2) can be 'true'
enable	condition(A_1) **enable** A_2 condition(A_1) **enable** condition(A_2)	*condition*(A_1)='true' $\longrightarrow$ *begin*(A_2) If *condition*(A_1)='true' then condition(A_2) will 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)	*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 1: Constructs for activity dependency specification in TAM

activities to be made available to other *concurrent* activities before end of the long-running activity, to increase the concurrency and avoid introducing the possibility of cascaded aborts. In this section we introduce two groups of activity restructuring operations, namely *activity-split* operations and *activity-join* operations, to allow users (or applications) to dynamically modify the set of concurrent activities while they are in progress.

3.1 A Motivating Example

Consider a development project (activity) T in computer-aided software engineering (CASE), which involves multiple tasks, requires data access from multiple remote sites, and has multiple designers cooperating in accomplishing the tasks. Suppose, to coordinate this collaboration-intensive project, the project manager breaks down the project into two main components: specification package S and implementation package P. Three software engineers are assigned to be responsible for specifications, say S_1, S_2, S_3, and four subsystems are planned for the implementation package, say P_1, P_2, P_3, P_4. At the initial stage the only interaction dependency realized is that the specification S should precede the implementation P (see Figure 2(a)). Suppose now both S_1 and S_3 have completed, and S_2 has been set aside for days because some important information required from a remote site is not available and S_2 has not been informed when the information will arrive. Due to such unpredictable situation, S_2 may take weeks or months to complete. The entire project may suffer from the following problems: (1) in case of system crashes at S, the finished work would get lost due to rollback at abort of S, (2) the subactivities of the implementation package which depend on only S_1 and S_3, have to wait for the entire activity S (i.e., S_2 in this case) to complete, even though they do not share resources with S_2. One obvious solution to these problems is to ungroup the activity S by a unnesting split, and associate the split operation with some additional activity dependencies between the resulting activities S_1, S_2, S_3 and other concurrent activities, such as P_1, P_2, P_3, P_4 in this case (see Figure 2(b)). Assume the following are the additional execution precedence dependencies (denoted by **precede**) added through the activity split: S_1 **precede** P_1, S_2 **precede** P_2, S_3 **precede** P_3, $\{S_1, S_2\}$ **precede** P_4. We can easily see that, with the restructured activity composition hierarchy in Figure 2(b), activities P_1, P_3 may begin as soon as S_1, S_3 are finished and committed and they do not have to wait for activity S_2 to complete.

This example demonstrates two common facts in enterprise-wide workflow applications: First, it is difficult to foresee the most effective clustering structure of activities at early stage of task breakdown. Second, many cooperative tasks may even not have a single clustering structure of activities that is effective at all time, because activities of both hierarchically complex structure and sophisticated interaction dependencies are more vulnerable to uncertainty and unforeseeable development. This is typically true when activities within a complex workflow process share resources, and some updated resources are retained from the others unnecessarily due to unpredictable development or accidental unavailability of data.

3.2 Activity-Split Operations

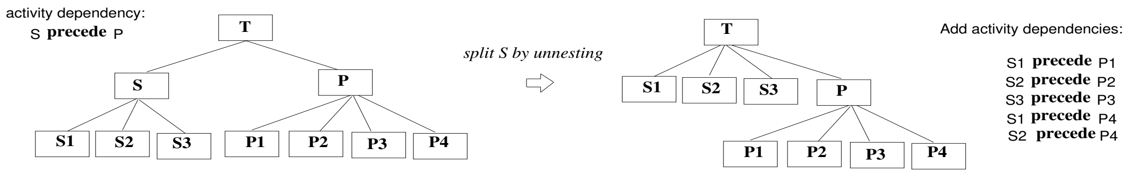
Given an ongoing activity, it is possible to split a leaf node or an internal node (including a root node) of the activity. The resulting nodes from the split could be mutually independent activities, or follow a serial execution order defined by the precedence dependencies, or be a set of activities whose execution orders are insignificant but whose states are dependent in one way or the other. The activity-split operations in general bring two major advantages into the transactional activities in progress:

- **adaptive recovery**, committing resources that are updated earlier and will not change.
- **added concurrency**, releasing the committed resources to allow access by other concurrent activities.

For simplicity of presentation, we describe two-way activity-split operations in this section. To split an activity n ways, we can apply the two-way split $n-1$ times in succession [14].

3.2.1 Two-way Activity-split Operations

An *activity-split* operation on an ongoing activity C, denoted by $ASplit(C, \varphi, C_1, C_2)$, produces two new activities C_1 and C_2 with a synchronization constraint $\varphi(C_1, C_2)$. It



(a) The initial task breakdown

(b) Dynamic split on S to deal with unpredictable or uncertain events

Figure 2: An application requiring dynamic restructuring of ongoing activities

Usage	Synchronization Constraint	Synopsis
$ASplit(C, p, C_1, C_2)$	**compatible**(C_1, C_2)	*parallel* activity-split operation (*p-Split*)
$ASplit(C, s, C_1, C_2)$	C_1 **precede** C_2	*serial* activity-split operation (*s-Split*)
$ASplit(C, pa, C_1, C_2)$	$commit(C_1)$ **disable** C_2	*parallel-alternative* activity-split operation (*pa-Split*) C_1, C_2 may begin together and abort C_2 if C_1 commits
$ASplit(C, sa, C_1, C_2)$	$begin(C_1)$ **precede** $begin(C_2)$ $\wedge$ $abort(C_1)$ **enable** C_2	*serial-alternative* activity-split operation (*sa-Split*) C_2 cannot begin unless C_1 aborts
$ASplit(C, cc, C_1, C_2)$	$commit(C_1)$ **precede** $commit(C_2)$	*commit-on-commit* activity-split operation (*cc-Split*) C_2 cannot commit unless C_1 commits
$ASplit(C, ca, C_1, C_2)$	$abort(C_1)$ **precede** $commit(C_2)$	*commit-on-abort* activity-split operation (*ca-Split*) C_2 cannot commit unless C_1 aborts
$ASplit(C, u)$	none	activity-split by *unnesting* (*u-Split*) if $Children(C) \neq \emptyset$, modify $Children(Parent(C))$ by $Children(Parent(C)) \cup Children(C)$, remove C

Figure 3: The group of activity-split operations in TAM

is possible to reuse the name of the original activity (C in this case) to name one of the resulting activities. The syntax for specifying synchronization constraint $\varphi(C_1, C_2)$ is the same as the one we use to specify activity execution dependencies in Section . For example, if the two resulting activities C_1 and C_2 can be submitted in parallel and can commit or abort independently of each other, then they are compatible to each other. We specify $\varphi(C_1, C_2)$ by **compatible**(C_1, C_2), and refer to such type of activity-split operations as *parallel activity-split* operation, denoted by $ASplit(C, p, C_1, C_2)$.

Figure 3 presents a group of specialized activity-split operators and their inherent semantics. Each of the specialized activity-split operators is associated implicitly with some synchronization dependency constraint and represents one typical or common case of activity splits. Figure 4 shows the specialization relationships between this given list of specialized activity-split operators.

3.2.2 Characterization of Activity-Split Operations

The *serial activity-split* (*s-Split*) operator splits an ongoing activity C into two activities that can be pursued and committed sequentially. It establishes a serial execution dependency between the two resulting activities (i.e., one cannot begin unless the previous one commits), which is captured by using the **precede** construct. The *serial-alternative split* (*sa-Split*) operator splits an ongoing activity into two alternative activities that are pursued and committed sequentially until one produces the desired result and commits. The serial-alternative split differs from the serial

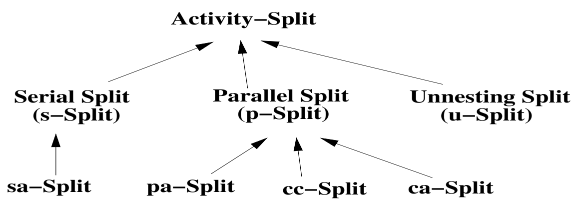


Figure 4: The specialization relationships among activity-split operators

split in the sense that one of the resulting activities only begins if the other one (the previous one) aborts.

The unique property of a parallel split is that the two resulting activities can be submitted and committed independently of each other. The *parallel-alternative split* (*pa-Split*) departs from the parallel split in the sense that the two resulting activities of $ASplit(C, pa, C_1, C_2)$ can be submitted together but they have an abort-on-commit dependency, that is, as soon as one commits the other will be aborted. The *parallel-alternative split* is a specialized version of *parallel split* that is quite common in practice. Similarly, the $ASplit(C, cc, C_1, C_2)$ (*cc-Split* for short) and $ASplit(C, ca, C_1, C_2)$ (*ca-Split* for short) operations can be seen as another two specializations of parallel split. The *cc-Split* results in two activities that have a commit-on-commit dependency, i.e., they can begin in parallel but one can commit only if the other commits first. The *ca-*

Split splits the ongoing activity C into two activities C_1 and C_2 satisfying a commit-on-abort dependency, that is, C_1 and C_2 can be pursued in parallel but C_2 can commit only if C_1 fails (aborts). The purpose of including *cc-Split* and *ca-Split* in our discussion is to use them as representative examples for the many other variations of parallel split that can be requested by the activity programmers using the generic form of activity-split operations $ASplit(C, \varphi, C_1, C_2)$.

In contrast to the serial or parallel activity-split operators, the unnesting split operator (*u-Split*) splits an ongoing activity by simply unnesting the activity hierarchy anchored at C. It means that if C has A as its parent and D, E, F as its child activities, $ASplit(C, u)$ would remove C from the set of child activities of A, denoted as $Children(A)$, by upgrading all children activities of C (D, E, F in this case) to become child activities of A, the parent activity of C, denoted by $Parent(C)$. Note that $ASplit(C, u)$ does not require any output parameters because it simply unnests the input activity C no matter how many children activities C may have. The *unnesting activity-split* is meaningful when used for splitting non-leaf node activities. It has zero effect when applying to a leaf node.

As some activity-split operators, when used to split a leaf node activity, may have different effects, regarding to its validity and its benefit, from the effects it may have when used to split an internal node activity, we below discuss the benefits and the validity of activity-split operations separately.

3.2.3 Benefits of Activity-Split Operations

In general, serial activity-split operations are applicable to both leaf node and internal node splitting. The main profit that a serial split brings into the transactional activities in progress is the additional adaptive recovery, that is, committing resources that are updated earlier and that will not change, thus avoid or reduce the overhead of cascaded aborts caused by activities higher in the hierarchy or system crashes. The difference (regarding benefits) between serial-alternative split (*sa-Split*) and serial split (*s-Split*) is more delicate in the sense that the *serial-alternative split* allows one to take advantage of the serial alternative characteristics to avoid or reduce the unnecessary executions or the executions that return null result. However, when using a serial split alone, either $ASplit(C, s, C_1, C_2)$ or $ASplit(C, sa, C_1, C_2)$, we would not be able to obtain added concurrency, because the serial split on a leaf node does not create more opportunity for other concurrent activities (sibling activities of C in this case) to see some partial result of C before the entire C finishes, although it has the benefit of allowing part of the results in C (C_2 in this case) to commit earlier so that when C aborts due to crashes C_2 is compensated rather than undo.

In contrast, the parallel split operation and each of its specializations, when applied legally, would bring both added concurrency and adaptive recovery into the ongoing activities. For instance, when applying to a leaf node that has a parent activity and several siblings, the parallel split operation $ASplit(C, p, C_1, C_2)$ results in two activities that can be submitted and committed in parallel (Figure 5(a)). Thus, after a parallel split, the sibling activities of the original activity C can access the resources updated by a subtask of C, for example C_1, without having to wait for the rest of C (C_2 in this case) to complete. Furthermore, the unexpected abort of C_2 would have no effect on the resources committed by C_1. In addition to all the benefits that a parallel split operation brings into the activity in progress, the parallel-alternative split operation $ASplit(C, pa, C_1, C_2)$ adds a further opportunity to reduce the duration of original activity C by allowing C_1 and C_2 to begin together and accepting only the one that commits first. In fact, most of the specialized parallel splits offer similar benefits as to the parallel activity-split. For example, the *cc-Split* operation and the *ca-Split* operation both allow the resulting activities to be pursued in parallel even though one resulting activity can commit if the other one commits (in case of *cc-Split*) or aborts (in case of *ca-Split*). In either of the cases, as soon as one resulting activity obtains the desired results and commits, its updates can be made available to other concurrent activities before the other resulting activity from splitting C terminates (commits or aborts).

3.2.4 Validity of Activity-Split Operations

An activity split operation is said to be *valid* if and only if the resulting activities are (1) TAM activities, i.e., satisfying the implicit activity dependencies implied in the activity composition hierarchy, such as the termination-sensitive dependency; (2) satisfying all the existing user-defined activities dependencies, and (3) they do not introduce any conflicting activity dependencies. In TAM we consider all valid activity splits as legal operations and all invalid activity splits as illegal operations.

Step 1: *The resulting activities from a split must be TAM activities*
The first step for verification of the validity of an activity-split operation is to check if the resulting activities are TAM activities. Figure 5 captures the effects of different ways of splitting a leaf node activity. The solid arrow between two activities denotes an activity composition relationship from parent to child, which inherently implies a *termination-sensitive* dependency: if the parent activity aborts, then all the active children activities are aborted, and all the committed ones are compensated. As mentioned earlier in Section , the termination-sensitive dependency (denoted by $\rightsquigarrow$) prohibits a child activity instance from having more than one parent, ensuring the hierarchically nested structure of activities. More precisely, there exists no such activities A, B, C in an activity hierarchy that $A \rightsquigarrow B \wedge C \rightsquigarrow B$ holds. This property is used extensively in verification of the validity of an activity-split.

For instance, a node splitting may introduce redundant dependencies or cause some existing subactivity to have more than one parent (see case (d) in Figure 5). To de-

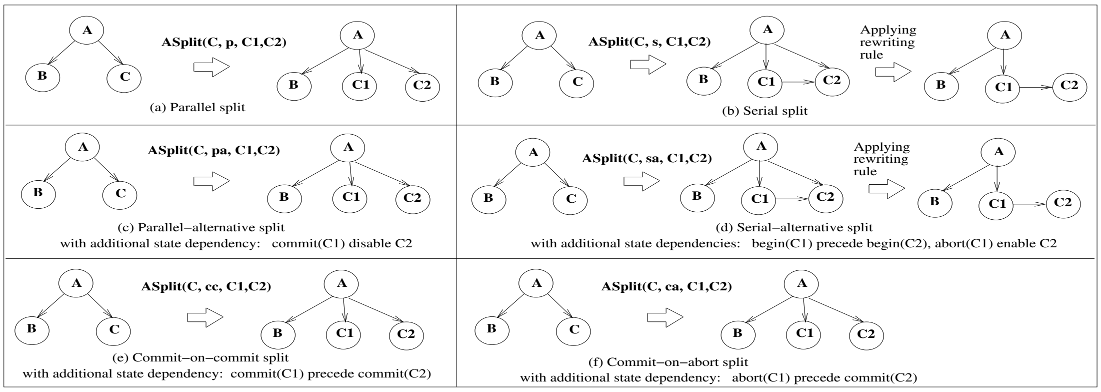


Figure 5: Splitting a leaf node

termine if the result of a split is legal, we first need to remove all the redundant dependencies before analyzing the legality of activity-split operations. The following consistency-preserving rewriting rule is used to remove all the redundant termination-sensitive dependencies introduced by activity-split operations:

$$A \leadsto B \wedge B \leadsto C \wedge A \leadsto C \iff A \leadsto B \wedge B \leadsto C.$$

By applying the above rewriting rule, we may conclude that the case (d) in Figure 5 does not violate the termination sensitive properties, thus the resulting activities are TAM activities.

When the input activity C is an internal node activity, only serial split and serial-alternative are legal split operations because the others would give rise to a violation of termination sensitive properties implied by the activity composition hierarchy. Figure 6 displays the effects of splitting an internal node activity using serial split, parallel split, and unnesting split. It shows that a parallel split on an internal node would result in a conflict with the activity composition hierarchy and the termination-sensitive dependencies, thus is considered as an illegal operation. Specializations of the parallel split, such as *pa-Split*, *cc-Split*, and *ca-Split*, are also illegal for the similar reason.

Step 2: *All existing activity dependencies must be semantically preserved after a split*
When a node C splits into two subactivities, say C_1 and C_2, the *termination-sensitive* dependencies between activity C and its parent activity A are assumed to be held (preserved) between C_1, C_2 and A. So do the subactivity execution dependencies and the activity state dependencies between C and its siblings, as well as the interleaving dependencies between C and children of its siblings. Thus the second step of the verification process is to conduct a dependency-preserving transformation. For example, suppose that the activity execution dependency "B **precede** C" holds before the split operation on C, then we would replace "B **precede** C" by

- B **precede** C_1 for $ASplit(C, s, C_1, C_2)$ or $ASplit(C, sa, C_1, C_2)$,
- B **precede** $\{C_1, C_2\}$ for the parallel split (*p-Split*) or one of its specializations, and
- B **precede** $Children(C)$ for the unnesting split.

A similar treatment is applicable to both the activity interleaving dependencies and the activity state transition dependencies. In TAM such transformation is provided as an integral part of the activity-split manager to guarantee that the resulting activities from each activity-split operation satisfy the set of existing activity dependencies that the original activity has.

Step 3: *The split operations should introduce zero conflict*
The last step of the validity verification is to ensure that all the additional dependencies introduced by the split operations do not bring in any contradiction to the set of existing activity dependencies. To automate this step it is important to know that the newly introduced dependencies by the split operations are syntactically correct and consistent with the set of existing activity dependencies. This is another reason that, in addition to support for the generic form of activity-split meta operation ($ASplit(C, \varphi, C_1, C_2)$), we also provide a list of specialized activity-split operators to add both the convenience to the users and the potential for performance improvement in the system implementation.

Proposition 1 Let $ASplit(C, \varphi, C_1, C_2)$ (φ-*Split* for short) denote a two-way activity split operation. The two-way activity-split operation $ASplit(C, \varphi, C_1, C_2)$ is *valid* if (1) C is a leaf node, or (2) $ASplit(C, \varphi, C_1, C_2)$ is a serial split operation or one of its specializations or a unnesting split operation when C is an internal node.

A formal proof of this proposition is omitted in this paper due to the space restriction. Readers may refer to our technical report [13] for detail.

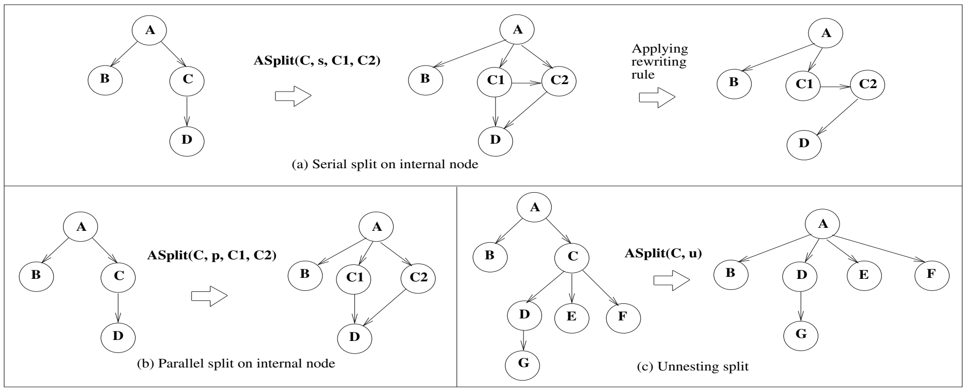


Figure 6: Splitting an internal node activity

3.3 Activity-join Operations

The inverse operation of activity-split, called *activity-join*, can combine results of several ongoing subactivities together and release them atomically, 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.

Similar to activity-split, for simplicity of presentation, we describe only the two-way activity-join. To join n activities we can apply two-way join $n-1$ times in succession. Because of the semantics of join (explained below), the *n-way* activity-join may be seen as syntactic sugar for successive two-way join.

Two types of activity-join operations are currently supported in TAM. The first type of activity-join operations is called "*join-by-group*", denoted by $AJoin(C, g, C_1, C_2)$. It groups two activities C_1 and C_2 by creating a new activity C as their parent activity, while preserving the activity composition hierarchy of each input activity. It is also called synchronous join. An activity-join by grouping is considered legal, if the two input activities C_1 and C_2 are sibling activities or independently ongoing activities. The second type of activity-join operations is referred to as "*join-by-merge*", denoted by $AJoin(C, m, C_1, C_2)$. It physically converges two activities in progress into a single activity, and also called asynchronous join. An activity-join by merge is considered legal, if the two input activities C_1 and C_2 are sibling activities, or one is a parent activity of another, or two independently ongoing activities, i.e., they do not have common parent activity. Figure 7 presents some examples of using activity-join operations for grouping activities that are at the same abstraction level or merging activities that are either at the same abstraction level or have a parent-child relationship.

For grouping or merging activities that are across different abstraction levels, a combination of split and join operations can be used. Note that we do not support for activity-join of arbitrary activities. For example, a call to join activity C with an activity that is a sibling activity of C's parent activity or C's grandparent activity is not acceptable in the TAM implementation. Our experience shows that the set of activity-join operations we support covers most of the practical requirements for grouping or merging ongoing activities in a cooperative group working environment.

To ensure the correctness criteria of activity-join operations, the merged or grouped activities must preserve all application-specific activity dependencies, such as subactivity execution dependencies, subactivity interleaving dependencies and user-defined state dependency dependencies. Due to the space limitation, we here omit the algorithms [13] for correctness ensurance of activity-join operations.

In summary, activity-split 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 adaptive recovery, added concurrency and enhanced cooperation, into the complex activities, and provide a cost-effective framework for organizing and restructuring complex activities in a distributed and cooperative computing environment.

To demonstrate the usefulness of the proposed framework, we have applied this activity restructuring framework to a number of example applications drawn from the real world domains. See [14] for further detail.

4 Related Work and Conclusion

Several activity models have been proposed [5, 4, 19] to support declarative specification of control flows within activities. Features of long running activities [5, 4] include an automatic compensation capability and the use of ECA-rules for monitoring activities. The cooperative model [19]

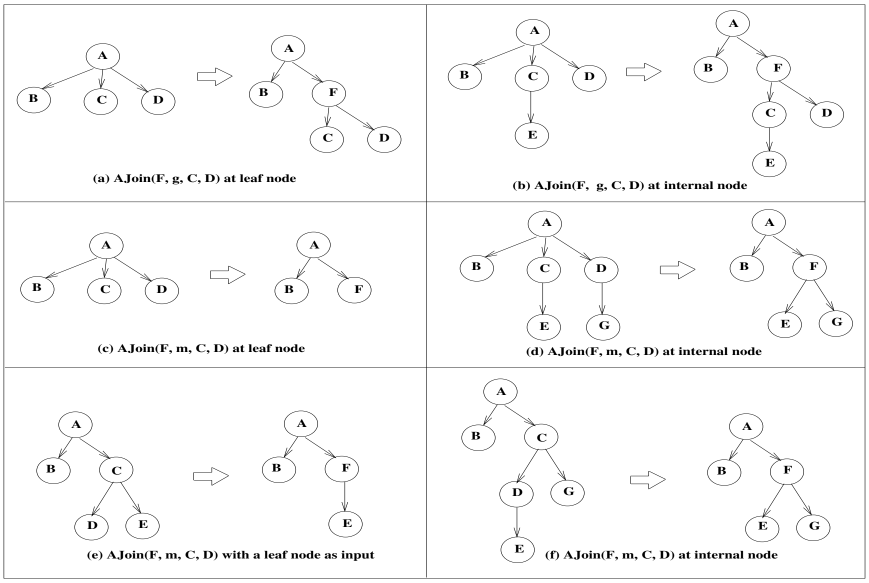


Figure 7: Examples of activity-join operations

achieves cooperation through controlled data exchanges in private workspaces and controlled sharing of a common database among users. Although previous activity models (e.g., [5, 4, 19]) and workflow systems (e.g., [9, 10, 15]) have been successful in capturing the business processes, they have limitations in the prediction of correctness of concurrent activities. Moreover, very few activity models or workflow systems proposed so far, to our knowledge, provide the adequate support for dynamic split and join of activities of deeply nested structures, ensuring the correctness of resulting activities.

A number of *extended transaction models* (ETMs) [6, 7, 8, 1, 16, 17, 18, 20] have been proposed, each targeting at a particular subset of the whole spectrum of interactions possible in advanced application environments. Therefore, an ETM alone is not sufficient for organizing complex cooperative activities that may require the whole spectrum of interactions [3]. Furthermore, most of the ETMs proposed so far are system supporting built-in ETMs, rather than system supporting user-defined ETMs, and thus not appropriate for supporting a variety of workflow applications with diverse or possibly conflicting business requirements [10, 15].

The TAM development is mostly inspired by the early work on split-transactions [18], Cooperative Transactions [1], and Transaction Groups [17]. Although cooperation among sibling transactions is supported by Cooperative Transactions or transaction groups [1, 18, 17], the interactions among siblings are either limited to static and one-shot design of transaction groups [17, 1], or restricted to leaf node transactions in order to support the isolation-based serializable split-transactions using read-sets and write-sets in a split call [18]. TAM differs from these models by three distinct features: (1) TAM activity specification language allows the users to specify activity composition hierarchy and activity dependencies declaratively and incrementally. Instead of restricting to interaction semantics among siblings, TAM captures interaction dependencies among activities across activity boundaries (e.g., interactions among child activities of different siblings). (2) To the best of our knowledge, no previous activity models have considered the transactional support for activity-split and activity-join operations to allow dynamic restructuring of activities that are hierarchically complex and have sophisticated interaction dependencies at the same time. (3) The third distinct feature is the systematic analysis of the benefits and validity of activity-split and activity-join operations. By specializing the activity restructuring meta operations, the efficient implementation of various split and join operations can also be explored.

The key contribution of this paper is its methodical development of a set of powerful activity-split and activity-join meta operations, such as *serial split*, *parallel split*, and *unnesting split* of an ongoing activity into two or more activities, and activity join by *merging* or *grouping* of two or more activities. An example of activity-split usage is for committing some work early or dividing on-going work among co-workers. An example of activity-join usage is the handing over of results to co-workers for integration into his/her own ongoing work. Furthermore, these activity restructuring meta operations can handle activities of both hierarchically complex structure and sophisticated activity execution dependencies, and preserve the correctness properties (as defined by TAM dependency specification) of new activities generated from split or join of ongoing activities. To demonstrate the expressive power and benefits of using our reflective framework for organizing and restructuring complex activities, we have outlined several applications drawn from telecommunications, collaborative design, and concurrent software development.

Our work on TAM continues along several directions. On the theoretical side, we are interested in strengthening our results on the assurance of correctness properties of concurrent activity execution despite the restructuring by activity-splits and activity-joins. On the practical side, we are looking into possibilities of building value-added adapters to support TAM on top of existing on-line transaction processing systems [2, 21] or other activity-based operational systems.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] A. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, Calif.), 1992.

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

[9] 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.

[10] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, (2):119–153, 1995.

[11] L. Liu and R. Meersman. The basic building blocks for modeling communication behavior of complex objects: an activity-driven approach. *ACM Transactions on Database Systems*, 21(3):157–207, June 1996.

[12] L. Liu and C. Pu. A transactional activity model for organizing open-edned cooperative activities. Technical Report TR96-21, Department of Computer Science, University of Alberta, An extended abstract in the Proceedings of the 31st Hawaii International Conference on Systen Sciences (HICSS-31) – Workflow Systems Track, Jan. 1998.

[13] L. Liu and C. Pu. Correctness issues for composition and restructuring of activities, Manuscript in Preparation, Oct. 1997.

[14] L. Liu and C. Pu. Methodical restructuring of complex workflow activities. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, March 1997.

[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] 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.

[19] 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.

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

[21] T. Zhou, C. Pu, and L. Liu. A practical method to implement the restructuring of transactional activities. Technical Report, Department of Computer Science and Engineering, Oregon Graduate Institute, July, 1997.