Dynamic Restructuring of Transactional Workflow Activities: A Practical Implementation Method

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

Abstract

We describe a practical method to implement the dynamic restructuring of distributed transactional activities with several main advantages. First, it guarantees the correctness of dynamic restructuring operations as well as that of ongoing transactional activities. Second, it is general enough to accommodate a wide range of workflow activities' dynamic restructuring requirements. Third, it facilitates the deployment of dynamic restructuring operations on production systems. Our method combines adaptors, microprotocols, and incremental specialization, building modularly on the system components (e.g., RTF [1], OCP [28], MARS [4]) that we previously developed to support distributed extended transaction processing [19]. A prototype implementation based on this method is built on top of Transarc Encina, a commercial online transaction processing (OLTP) monitor. To our knowledge, this is the first implementation of dynamic restructuring on top of commercial OLTP systems.

1 Introduction

Many of today's business workflows execute in distributed environments, exhibit dynamic structure, and involve sophisticated interaction dependencies, making them vulnerable to uncertain duration and unpredictable development. Handling dynamic transactional workflows in practice is considered one of the challenging open problems in workflow research [24]. In order to optimize the structure of business workflows and their execution performance in anticipation of various uncertainties, we have introduced a family of operations (e.g., Activity-Split and Activity-Join) for dynamic restructuring of ongoing workflow activities [16]. These activity restructuring operations build on a formal notion of validity so that they guarantee a rich set of correctness criteria (that extend serializability) of the resulting distributed, dynamic, and sophisticated new activities. Despite the conceptual simplicity of these activity restructuring operations, their implementation on practical systems is challenging for a variety of reasons: (1) these restructuring operations need to be carried out dynamically and efficiently while activity instances are executing; (2) activity instances have transactional properties to be met in addition to their normal execution dependencies, even in presence of these restructuring operations; (3) these restructuring operations should work as well on distributed workflow activities; and (4) a practical implementation should build on existing software to facilitate integration into production systems.

In this paper, we strengthen the case for transactional workflow activities by describing a practical method to implement the dynamic restructuring of distributed transactional activities. The method offers several main contributions. First, the resulting implementation guarantees the validity requirements of dynamic restructuring operations as well as the correctness of ongoing workflow activities, and is flexible enough to accommodate a variety of these activities' transactional semantics. Second, our method is general enough to support a wide range of workflow activities' dynamic restructuring requirements, such as instance or group based restructurings. Third, our method facilitates the deployment of dynamic restructuring operations on production software systems. The method divides the restructuring operations into *microprotocols*, and builds *adaptors* on top of commercial transaction processing (TP) systems. Microprotocol composition further facilitates the adoption of *incremental specialization* [21] to create specific and efficient instances of activity restructuring operations. Our method also builds modularly on the system components (e.g., RTF [1], OCP [28], MARS [4]) that we previously developed to support distributed extended transaction processing [19].

To demonstrate the method, our additional contribution is an implemented prototype system on top of Transarc Encina, a commercial online transaction processing (OLTP) monitor. To our knowledge, this is the first implementation of dynamic restructuring of transactional activities on top of commercial TP systems. Our implementation not only demonstrates the feasibility of providing transactional restructuring functions in commercial systems, but also illustrates how one can implement such functionalities as an integral part of a new system, or as an add-on to an existing system.

There are several other aspects of transactional workflow activities that are not addressed in this paper. For example, the recovery issues in association with the dynamic restructuring operations are addressed in a separate paper [27]. Other examples include further formal development of transactional activities, exploration of activity composition, and interoperation of autonomous and heterogeneous transactional workflow systems. Nevertheless, we believe that our implementation method constitutes an important building block of and takes another concrete step towards making practical transactional workflow activities into reality.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of transactional activities and the functionalities of the dynamic activity restructuring operations. We elaborate our practical method to implement these restructuring operations in Section 3. Section 4 describes our prototype system that supports transactional activities and these dynamic activity restructuring operations. Related work are reviewed in Section 5, and Section 6 concludes the paper. Because of the length restriction, we refer interested readers to [29] for further technical details.

2 Dynamic Activity Restructuring Primitives

2.1 Transactional Activity Model

We first describe a general activity model as our basis for discussion. This model encompasses features from previously published activity models [22, 3, 15].

Activities are structured programs that exchange information with other activities, databases, files, and users. In our base model, we distinguish between parameterized *activity specifications* and their run-time instantiated *activity instances*. We also assume activities are structured, and distinguish therefore between *elementary activities* and *composite activities* with a tree organization which we call *activity composition hierarchy*. We sometimes refer to the component activities of a composite activity as *constituent activities*.

The activity composition hierarchies capture the structures in business applications, and implicitly capture the *transactional dependencies* caused by concurrent data sharing and failure recovery as well. For example, consider an electronic purchase activity between a customer and an online-merchant, which has several constituent activities: deal-offering, payment, and goods-delivering. The payment activity in turn has its own subactivities, like pricing and generating receipt. The activity composition hierarchies not only captures this business structure, but also captures the business process's transactional requirements like: if the payment does not succeed, the complete purchase activity has to be aborted. Or, the payment activity can commit even if the receipt is not properly produced. The special treatment for transactional dependencies compared with normal execution dependencies relieves process designers from explicitly specifying the actions that deal with concurrent synchronization and failure recovery, which are complex and might not be familiar to the designers. Besides activity composition hierarchy, our model should also provide a set of constructs for users to incrementally and declaratively specify sophisticated activity dependencies that are application-specific, like those dependencies during an activity instance's normal execution. In above example, the programmer could specify different schemes used to carry out the online-purchase: pay before delivery, incremental payment, pay after delivery, etc.

When specifying execution dependencies among activities, we are particularly interested in those activity states that are externally observable and those state transition dependencies that can be externally controlled. For example, `Initialized`, `Active`, `Subcommitted`, `Commit`, `Abort`, and `Compensated`. These are abstract states encapsulating actual program states that comprise of variable values. Activity states are related by *state transition dependencies*. A state transition dependency constrains activities' entering of specific states. Still in above electronic purchase scenario, (`Commit`(deal-offering) && `Commit`(payment)) **precede** `Commit`(goods-delivering) could represent an activity dependency where the goods-delivering activity can not enter the `Commit` state when either the deal-offering or payment activity has not entered its `Commit` state.

2.2 Dynamic Restructuring Primitives

A two-way *Activity-Split* operation on an ongoing activity instance C, denoted by $ASplit(C, \varphi, C_1, C_2)$, produces two new activity instances C_1 and C_2[1]. The synchronization constraint φ is a function of (C_1, C_2) that captures the dependencies between them after the split. For example, suppose C_1 and C_2 are independent of each other, we refer to such type of Activity-Split operations as *parallel Activity-Split*, denoted by $ASplit(C, p, C_1, C_2)$. N-way Activity-Split can be easily constructed using the two-way Activity-Split [16]; and for presentation clarity, we restrict our discussion to two-way Activity-Split in this paper.

An Activity-Split operation is *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 transactional dependency; (2) satisfying all the existing user-defined activity 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.

Activity-split is useful in several ways. For example, it supports adaptive recovery from permanent or temporary unavailability of nodes, users, or constituent activities. Another benefit is added concurrency, by releasing early-committed resources through a split activity to allow accesses by other concurrent activities.

The inverse operation of Activity-Split, called *Activity-Join*, combines results of several ongoing subactivities together and releases them atomically, as if they had been a single activity. Activity-Join has two specialized versions. The first type of Activity-Join operation 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 *valid*, 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 *valid*, 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.

In summary, Activity-Split and Activity-Join can be combined in any formation. By allowing the release of early-committed resources or ownership transfer of uncommitted resources, these dynamic activity restructuring operations bring into complex activities a number of advantages, such as adaptive recovery, added concurrency, dynamic process evolution, and enhanced cooperation, providing a cost-effective framework for organizing and restructuring complex activities in a distributed and cooperative computing environment.

[1] It is possible to reuse the name of the original activity C to name one of the resulting activities.

3 A Practical Implementation Method

3.1 Overview of Method

The first building block of our implementation method is the careful design of the dynamic activity restructuring to facilitate modular, adaptable, and efficient implementation. We achieve this by combining microprotocols [18], open implementation [13], and incremental specialization [21].

We have identified a number of key issues that need to be addressed by an implementation of dynamic activity restructuring, such as: how to control an activity instance's execution under the possible influence of restructuring at run-time; how to deal with an activity instance's transactional scope changes; how to enforce activity execution dependencies and how to detect/resolve dependency conflicts due to restructuring; how to handle the case when restructuring touches distributed activities, and so on. For each of these issues, we provide our own solutions in a single or a suite of microprotocols. For each microprotocol, we then carefully design two types of interfaces: *functional interface* and *meta interface* to accommodate a wide range of workflow activities' restructuring needs. We also employ incremental specialization techniques to build restructuring primitives that are optimized with respect to special application situations.

The second building block of our implementation method is the system components developed in our previous effort on supporting distributed extended transactions [19]. In particular, the Reflective Transaction Framework (RTF) supports different extended transaction models on top of production transaction processing (TP) systems; the Open Coordination Protocol (OCP) is a flexible coordination facility for building different distributed coordination protocols for distributed extended transaction processing. By utilizing and extending these system components, we guarantee the transactional properties of distributed workflow activities in presence of the activity restructuring operations.

The third building block of our implementation method is a set of transactional activity management adapters (*TAM adapters*), which are *add-on* modules to RTF, OCP, or an existing transaction processing (TP) monitor. Each TAM adapter encapsulates and extends the functionalities of its underlying layer. Like the *transaction adapters* in RTF, TAM adapters are based on the commands and functionality of the well-documented TP monitor reference architecture [10] for practicality. Furthermore, our implementation of the TAM adapters relied only on a small but widely supported set of API in the TP monitor reference architecture.

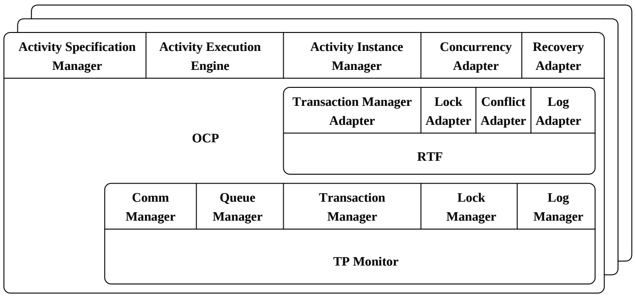

Figure 1: Transactional Activity Processing Framework

Figure 1 illustrates the relationships among TAM adapters, RTF, OCP, and TP monitors in a distributed setting. Each outer-most bounding frame in the figure represents an entire transactional activity processing component on one distributed node. Within each component (i.e., each frame), three layers are present: TP monitor layer, RTF and OCP layer, and the TAP layer. OCP and the transaction adapters (e.g., transaction manager adapter, lock adapter, etc.) in RTF utilize and extend the functionality of TP monitors to handle distributed extended transactions. TAM adapters utilize and extend the functionalities of RTF, OCP, and TP monitors to handle dynamic transactional activities. Together, these adapters and the underlying TP facility form a framework for dynamic transactional activity processing.

By applying the above implementation method, activity restructuring primitives are composed of a suite of microprotocols and invariants/quasi-invariants. The microprotocols utilize RTF, OCP, and the TAM adapters in the transactional activity processing framework. The invariants and quasi-invariants guide the specialization process at primitive instantiation time to build efficient primitive instances. Furthermore, the adapters make our implementation easy to build on top of production TP systems.

3.2 A Walkthrough Example

Here, we describe an example transactional activity on telecommunications that we will frequently refer to in our rest implementation descriptions.

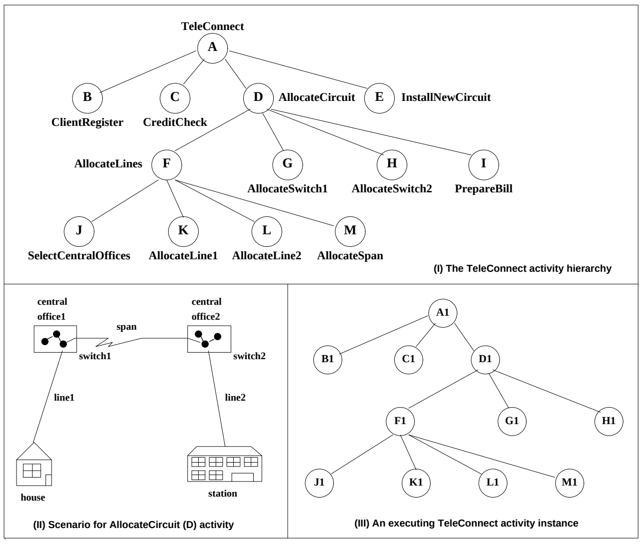

Figure 2: An Example Telecommunication Application

Consider a composite activity `TeleConnect` that performs telephone service provision task by installing and billing telephone connections for a telecommunication company. Suppose the activity A:`TeleConnect` consists of four subactivities B:`ClientRegister`, C:`CreditCheck`, D:`AllocateCircuit`, and E:`InstallNewCircuit`. A is initiated when a customer requests telephone service installation. Activity B registers the client information in the client database for billing purpose and passes a service order to activity C. Activity C evaluates the credit history of the client. Activity D attempts to provide a connection by allocating lines (F:`AllocateLines`), allocating slots in switches (G:`AllocateSwitch1`, H:`AllocateSwitch2`), and preparing bill for establishing the connection (I:`PrepareBill`). The activity of allocating lines (F) in turn has a number of subtasks, such as selecting nearest central offices (J), relocating existing lines (K:`AllocateLine1`, L:`AllocateLine2`) and spans (trunk connection) between two allocated lines (M:`AllocateSpan`). The activity E:`InstallNewCircuit` is designed to perform an alternative task that involves physical installation of new facilities in the event activity D fails. Figure 2(I) presents the complete activity hierarchy for `TeleConnect`; Figure 2(II) illustrates the scenario for D:`AllocateCircuit`; and Figure 2(III) shows the hierarchy of a particular running instance $A1$ of the `TeleConnect` activity. A, D, and F are composite activities, while all the rest are elementary activities. Also note that the figures here do not particularly show activities' normal execution dependencies, like serial dependency. For space limitation, we do not include the complete activity specification [29] here; however, we would explain relevant dependencies as such where necessary in the following sections.

3.3 Activity-Split Implementation

Now we present a detailed implementation description of Activity-Split. As mentioned in Section 2, we only illustrate the implementation with that of two-way Activity-Split.

A two-way Activity-Split operation splits an activity instance A into two activity instances A_1 and A_2. In order to ease the specification task and apply Activity-Split incrementally, we assume the operations being split from a composite activity (CACT) A can only be A's children activity instances, but not arbitrary descendant activity instances. For example, in Figure 2(III), we can use Activity-Split to split activity instance F_1 from D_1, but we can not use it to split D_1's descendant J_1 from D_1 directly. For activity splits that involve activity instances across different hierarchical levels, a combination of the Activity-Split operations are used. To split an elementary activity (EACT) instance, the `Split-transaction` operation [20] will be used as Activity-Split. The following is the C declaration for the two-way *instance-level* Activity-Split on a CACT:

```
status_t Activity-Split(
    Aid_t *Original_A_id,
    Aid_t_list *A1_child_list,
    Aid_t_list *A2_child_list,
    Dependency_list *A1_A2_depen_list,
    boolean_t ASplit_tag
)
```

In above function declaration, `Original_A_id` is the identifier of the activity instance that is being split. The two subsequent arguments identify those activity instances that are to become the children of the two new activity instances resulting from the Activity-Split operation. The fourth argument `A1_A2_depen_list` is a list of new activity dependencies between the two new activity instances. The last argument is a tag indicating whether a specialized Activity-Split instance would be used, like whether the two after-split activity instances execute in parallel or sequentially. It is used by the system to select a built-in primitive and choose the right set of default activity dependency translation rules (see "Instance Split").

Example 3.3.1 *This example would be referenced in the rest of the paper.*
For the running `TeleConnect` instance in Figure 2(III), we apply the following Activity-Split restructuring operation: supply D_1: `AllocateCircuit` as the `Original_A_id` parameter, F_1: `AllocateLines` as the only child instance in `A1_child_list`, $\{G_1$: `AllocateSwitch1`, H_1: `AllocateSwitch2`$\}$ as `A2_child_list`, and enforce an extra dependency "`Commit`(A2) **precede** `Commit`(A1)" between A1 and A2. We also require this particular Activity-Split operation to be parallel, with the parameter `ASplit_tag` being `PARALLEL`.

Dynamic Execution Control

A key feature of dynamic activity restructuring operations (and dynamic activities in general) is that they are *dynamic*, i.e., they can be invoked while activity instances execute at run-time. This in consequence brings a set of questions for any implementation of these restructuring operations to answer, such as: whether an instance would be affected by a restructuring, how we enable an instance to adapt to the specification changes caused by a restructuring, how it can adapt to these changes without losing its current execution context, and so forth.

The Dynamic Execution Control microprotocol deals with these questions. Our solutions here comprise of several parts: First, we propose a two-level control architecture with flexible concurrency control policies at both levels. At the *specification-level*, the Activity Specification Manager controls accesses and updates to activity specifications, which are stored in the `Activity Specification Table` and the execution `Dependency Graph`. At the *instance-level*, the Activity Instance Manager controls activity instance contexts, stored in the `Activity Instance Table`. The `Activity Specification Table` records activity composition hierarchy information and references the `Dependency Graph`, which stores activity execution dependencies (Section 2). Our added specification-level control provides running activity instances with the flexibility to adapt to changes in their specifications. Each activity instance then declares its change adaptation intentions, such as: do not follow changes, restart and use new specification, adapt to changes in the current execution context, etc.

For the `TeleConnect` activity hierarchy shown in Figure 2, Table 1 illustrates the content of the `Activity Specification Table` for all the involved activities. And Table 2 illustrates the content of the `Activity Instance Table` for a running instance (A_1) of the `TeleConnect` activity.

Activity	Superior	First Child	Next Sibling	...
A	NIL	B	NIL	...
B	A	NIL	C	...
C	A	NIL	D	...
D	A	F	E	...
...	...	...	...	...

Table 1: Activity Specification Table

Activity Identifier	Activity State	Activity Specification	Superior Identifier	First Child	Next Sibling	...
A_1	Active	A	NIL	B_1	NIL	...
B_1	Commit	B	A_1	NIL	C_1	...
C_1	Commit	C	A_1	NIL	NIL	...
D_1	Active	D	A_1	F_1	NIL	...
F_1	Active	F	D_1	J_1	G_1	...
...	...	...	...	...	...	...

Table 2: Activity Instance Table

An question associated with above is the activity execution model, i.e., how activity execution dependencies are enforced. Our second design component here is a hybrid model with both OS-execution and dependency graph interpretation. The reason is to avoid the overhead of run-time code generation or graph interpretation at the specification-level, and at the same time to avoid the static nature of

OS-based process/thread execution (with respect to adapting to source code changes). Each activity instance uses checking routines before and after each activity state change (abstract and external state, Section 2) with the Activity Specification Manager, as synchronization points to detect (and then adapt to) specification changes. In between these state changes, the activity instance's execution is taken care of by an OS.

A third component here is the adoption of server-based activity scheme, in that each activity instance can run in a separate process space, as in a distributed setting. This is chosen to allow dynamic binding of activity instances whose specifications were not present in the original activity specifications. Such cases can easily arise in today's business environment because of unpredictable process structure or even exception handling actions for a predefined process model.

Specification Split

For a two-way Activity-Split operation that splits activity instance A_1 into B_1 and C_1, this microprotocol "splits" A_1's corresponding specification A into two new specifications, one for B_1 and another for C_1. Such "specification split" is performed on the `Activity Specification Table` that is shared by all running activity instances of A, unless indicated as a per-instance case by A_1. In the latter case, the Activity Specification Manager makes a copy of A's relevant entries from the `Activity Specification Table` for A_1 and performs this microprotocol only on that copy.

Generally, the microprotocol carries out the following four-step operations. In the discussions, we assume activity instance D_1: `AllocateCircuit` is split into D_2 and D_3 in **Example 3.3.1**, with F_1: `AllocateLines` becoming D_2's only child instance.

First, all the specification entries relevant to the current Activity-Split operation are locked in exclusive mode, to prevent "change-obliged" activity instance(s) from reading and following incorrect specifications. those specifications and continue executing In our example, this microprotocol locks entries A, D, F, G, and H.

Second, we create two new specification entries representing the two after-split activities'. And, we fill in the content of these two new entries with relevant information from Activity-Split's parameters, e.g., a list of children activities and dependencies among them. In **Example 3.3.1**, we create two new entries D_α and D_β corresponding to D_2 and D_3. Then we fill in D_α's entry with its only child activity F, registering their parent-child relationship; and we fill in D_β's entry with its children activities G, H, and I, also registering their parent-children relationships.

Third, we translate the hierarchy-related transactional dependencies imposed on the splitting activity into those imposed on the two after-split activities. This is done by the Activity Specification Adapter's modifying fields `Superior Activity`, `First Child Activity`, and `Next Sibling Activity` in relevant specification entries. In **Example 3.3.1**, the new specification entries of A (private to activity instance A_1) will look like those in Table 3, in comparison with the original specification entries in Table 1.

Next, for each activity dependency that is imposed on the splitting activity, we translate it into a set of dependencies that are imposed on the two after-split activities[2], by applying a set of dependency-specific translation rules that are recorded in the `Dependency Translation Table`. The most important type of dependency in this step is activity execution dependency (static) that includes intra-activity control flow and data flow dependencies. Also note that the activity dependencies translated in this step are mostly for those *unexecuted* constituent activities, and thus does not include dynamic activity dependencies caused by concurrent data sharing, which will be dealt with in the "Instance Split" microprotocol.

[2]We separate the translation of the transactional dependencies from those in this step, because the former is critical to preserving activities' transactional properties and requires special treatment. We also dedicate special fields in relevant data structures to speed up processing such dependencies.

Activity	Superior	First Child	Next Sibling	...
A	NIL	B	NIL	...
B	A	NIL	C	...
C	A	NIL	D	...
D	A	F	E	...
D_α	A	F	D_β	...
D_β	A	G	E	...
F	D_α	J	G	...
G	D_β	NIL	H	...
...	...	...	...	...

Table 3: Activity Specification Table 2

Instance Split

For a two-way Activity-Split operation, this microprotocol actually splits an activity instance into two new activity instances, by performing operations on relevant entries in the `Activity Instance Table`. The logic is quite similar to that of Specification Split so we omit much of the similar descriptions here. For **Example 3.3.1**, two new entries D_2 and D_3 are created in the `Activity Instance Table`. The new instance entries of A_1 will look like those in Table 4, in comparison with the original instance entries in Table 2.

Activity Identifier	Activity State	Activity Specification	Superior Identifier	First Child	Next Sibling	...
A_1	Active	A	NIL	B_1	NIL	...
B_1	Commit	B	A_1	NIL	C_1	...
C_1	Commit	C	A_1	NIL	D_1	...
D_1	Active	D	A_1	F_1	NIL	...
D_2	Active	D_α	A_1	F_1	D_β	...
D_3	Active	D_β	A_1	G_1	NIL	...
F_1	Active	F	D_2	J_1	G_1	...
G_1	Active	G	D_3	NIL	H_1	...
...	...	...	...	...	...	...

Table 4: Activity Instance Table 2

For Activity-Split, the system has several sets of default dependency translation rules, one for each specialized dynamic restructuring primitive built into the system. For example, depending on whether the two after-split instances execute in parallel or sequentially, the system will use the corresponding set of rules. This microprotocol also provides meta interface for customizing these dependency translation rules.

An important dependency type, which is handled by this microprotocol but not by the "Specification Split" microprotocol, is what we call *dynamic activity dependencies* that are caused by activities' concurrent execution with shared data. Such activity dependencies are generated and processed by a special Activity Concurrency Adapter plus the relevant adapters in RTF (e.g., Lock Adapter and Conflict Adapter) [1] that together are capable of carrying out both traditional and advanced concurrency control (CC) algorithms (e.g., semantic CC, divergence control, etc.). Once generated, these dependencies are also stored in the `Activity Instance Table` at run-time for relevant activity instances; so this microprotocol just uses such information as is — not much different from the other types of dependencies. Besides performing the correct translation, there is no additional processing required for this microprotocol to deal with dynamic activity dependencies caused by activities' data sharing.

Dependency Addition and Conflict Resolution

After above microprotocols, all the existing dependencies imposed on the splitting activity instance have been translated into a proper set of dependencies that are imposed on the after-split activity instances. Nevertheless, there might be additional dependencies that are passed as parameter (i.e., the `A1_A2_depen_list` parameter) to Activity-Split. For instance, the additional commit precedence dependency between D_2 and D_3 (i.e., the two after-split instances) in **Example 3.3.1**.

For the two after-split activity instances, this microprotocol adds these "additional" dependencies to those already obtained by applying the dependency translation rules in previous microprotocols. While doing this, the microprotocol checks for potential conflicts between these new dependencies and those existing ones. In **Example 3.3.1**, if there is a serial dependency between G and H in the original specification, upon instance D_1's split, such dependency is preserved and mapped onto G_1 and H_1. But if in the parameter to Activity-Split, we force a parallel-dependency between G_1 and H_1 (e.g., by turning each into the child of a different instance), this will conflict with the existing serial-dependency.

The microprotocol detects such conflicts using a rule-based method. The TAM system has a set of built-in conflict detecting rules. These rules are assumed to be *correct*, but not necessarily *complete*. The system allows customizations of these built-in rules and additions of new rules via this microprotocol's meta interface. The agent who makes such customizations or additions is responsible for ensuring the consistency between the new rules and existing rules, so that the rules themselves do not conflict. Besides conflict-detection, the TAM system will also ask the user for conflict resolution strategies after conflicts occur, whether the conflicts are activity dependency conflicts or detection rule conflicts. After all such conflict resolutions are done, the splitting activity instance's appearance in all relevant data structures are eliminated.

Note that the dependency conflict resolution in general is an NP-hard problem. Therefore, TAM adopts an approach similar to those in [11, 2]. Specifically, we only check dependency conflicts for a restricted class of dependencies, which is built into the TAM system. This microprotocol would reject new activity dependencies that conflict with this built-in class but may be conflict-free under other interpretations. For customized dependency types, the application that makes that customization is solely responsible for the correctness of the dependency consistency checking.

Distribution Control

When activity instances involved in a dynamic restructuring are distributed, we might need to use a distributed protocol to coordinate them to guarantee the correctness of the restructuring operation. For such, we use the Open Coordination Protocol (OCP) [28] to build appropriate coordination protocol instances. Figure 3 and 4 illustrates a simple two-phase Activity-Split composition, with `Preparation` and `Action` being the interfaces of the two phases.

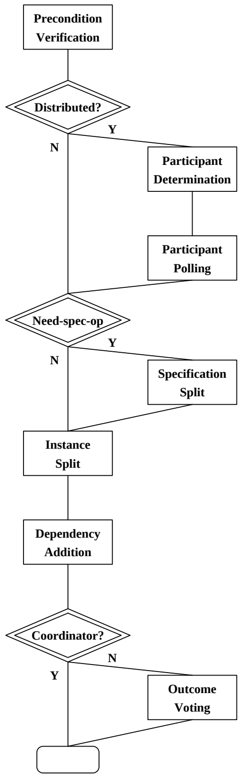

Figure 3: Microprotocol composition in Activity-Split `Preparation`

3.4 Activity-Join Implementation

The microprotocol composition for Activity-Join is very similar to that for Activity-Split. For the lack of space, we refer the readers to the longer version of this paper [29] for details.

4 System Prototype

We have implemented a prototype system for transactional activity management. The system covers a family of dynamic activity restructuring operations, as well as other important features of the Transactional Activity Model (TAM) [15]. The implementation of the activity restructuring operations exactly followed our method described in previous section. The complete system is client-server oriented. The backend server features a set of transactional activity management primitives, and is built on Transarc's transaction processing (TP) monitor Encina[3]. The frontend features several web-friendly graphical user interfaces (GUIs) for activity control and administration at both specification and instance levels, and is built with a mixed use of Java applets, JavaScript, and HTML for portability and reusability. Currently, the frontend and backend server communicate via some CGI gateway programs that mediate between HTTP and Encina TRPC (transactional RPC based on DCE-RPC). We have also developed a transactional application `TeleConnect`, as described in Section 3.2, that takes advantage of the functionalities our prototype system provides.

[3]Currently, we only have Encina for Sun-Solaris platforms; so the backend server is only built on these platforms.

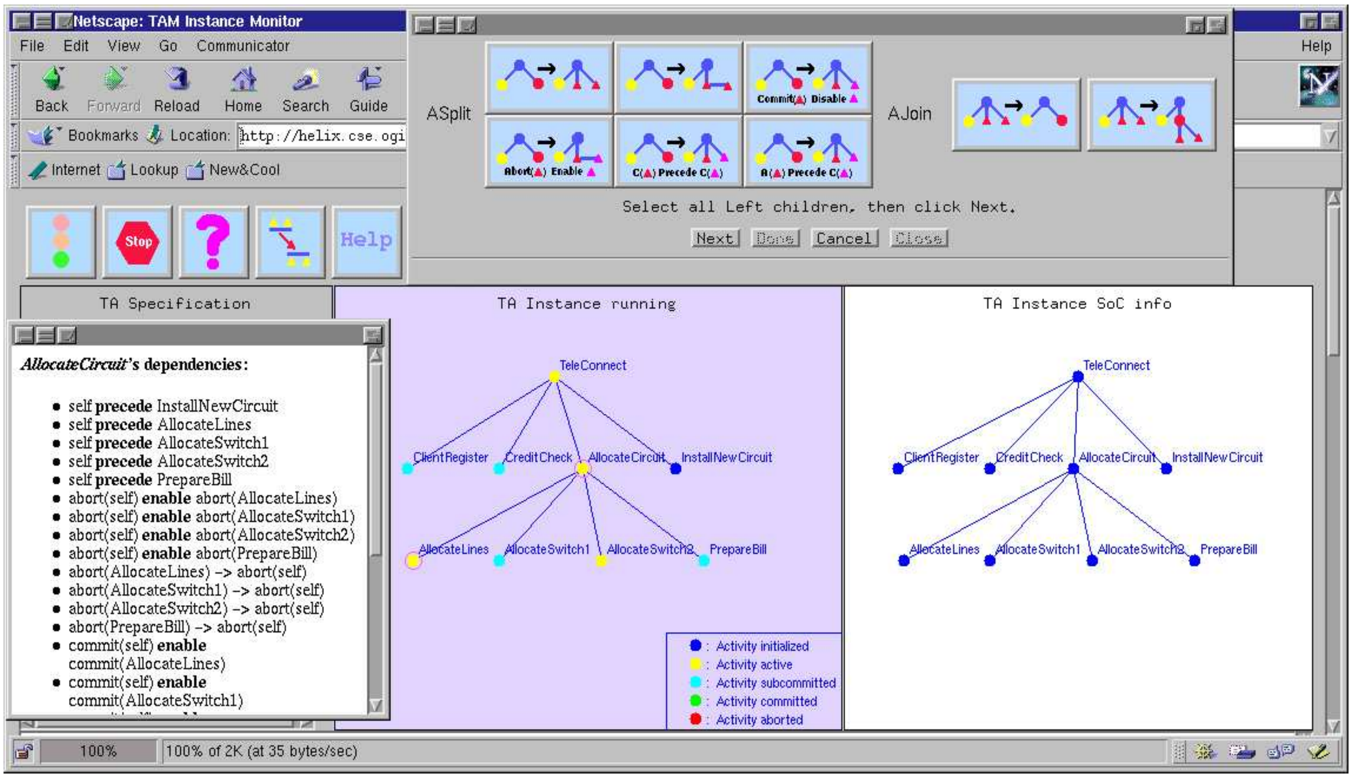

Figure 5: A screenshot of our prototype system

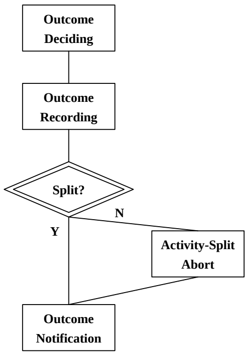

Figure 4: Microprotocol composition in Activity-Split Action

5 Related work

Recent years have seen a wide range of commercial workflow products (e.g., FlowMark, InConcert, Lotus Notes, etc.), which primarily aim at automating document-driven business processes. The functionalities of these products are currently too limited to effectively support enterprise-wide workflows [9, 24]. For example, they may not ensure workflow correctness in presence of concurrency or failures. Some of these products allow dynamic modifications of workflow process specifications [9]. However, the lack of rigorous transactional guarantees in these products inherently limits the value of such functionality; yet also, these support only deal with specification-level changes with simplistic policies [2], which is not flexible enough to support the wide range of activity restructuring needs. For example, the problem of instance-specific activity restructuring, which frequently arises due to unexpected situations during each activity instance's execution, is left unaddressed. In comparison with the commercial effort, transactional workflow (TWF) research [7, 25, 22, 26, 17] aims at providing transactional support for business workflow processes. But to date, few have adequately addressed the issue of handling dynamic transactional workflows, either [22, 3]. Managing dynamic workflow activities with rigorous support for transactional guarantees (e.g., general recoverability, flexible concurrency constraints) remains an important open problem in both research and industry sectors [9, 24]. We believe we have made a concrete step towards providing such dynamic restructuring support on production systems.

Our dependency handling component adopts a temporal logic based approach, similar to that in [11]. Because of the execution model we assume, our dependency management policy is more active in comparison. Rather than purely waiting for activity instances to request state change condition checkings, our dependency component might actively invoking activity instances that are involved in a specification change.

There have been several work on transaction restructuring, like chopping transactions [23, 12] and split/join transactions [20, 1]. But they either are statically done at compile-time, or only work on flat structures. In comparison, our work supports dynamic activity restructuring with transactional considerations, and work on structured activities.

On one hand, our implementation method benefits from several system development techniques and principles that have been demonstrated successfully in other fields like operating systems and programming languages. The idea of microprotocol decomposition comes from the x-kernel work [18] on building flexible network protocols. Specialization was proposed by the Synthetix project [21], and is closely related with partial evaluation [6] in programming language research. Our approach to system extensibility continues to follow the Open Implementation (OI) principle [13] as in [1, 28]. On the other hand, our implementation method and prototype system leverage and build directly on our own experience and mechanisms. Specifically, our transactional activity processing framework builds on RTF [1], OCP [28], and MARS [4], which systematically extend commercial TP monitors to make the implementation of a wide range of distributed extended transaction models practically available [19].

6 Conclusion

Workflow systems are an important part of many business applications today, particularly in the capture and execution of business processes. The execution of workflows are usually straightforward, without considering alternative executions. Dynamic restructuring operations can help a workflow system bypass temporary or permanent bottlenecks in both data flow and control flow of workflow processes, like those caused by node/network unavailability, user intervention or absence, and other exceptions.

In this paper, we have presented a practical method to implement dynamic restructuring operations for distributed transactional activities. The method modularly decomposes the restructuring operations into microprotocols, and maps them into the system components (RTF [1], OCP [28], MARS [4]) we previously developed to support distributed extended transactions [19], as well as a set of adaptors on top of production transaction processing (TP) monitors. Incremental specialization [21] is applied to improve operation performance. Our method is general enough to accommodate a wide range of workflow activities' characteristics and dynamic restructuring needs (e.g., dynamic requirements, transactional requirements, dependency enforcement requirements, etc.), and provides valuable insights into how dynamic activity restructuring can be practically supported on production systems.

Based on above method, we have implemented a prototype system on top of Transarc's transaction processing (TP) monitor Encina. Our current system covers a family of dynamic restructuring operations from [16], as well as many other features of the Transactional Activity Model (TAM) [15]. To our knowledge, this is the first implementation of dynamic restructuring on top of commercial TP systems. We hope our work will encourage and lead to more implementations of transactional activity restructurings in commercial systems.

This paper captured just another step forward in the practical support for adaptive transactional activities. We are currently using our prototype system to evaluate the performance of these activity restructuring operations and the benefits they bring to workflow activities.

References

[1] R. S. Barga and C. Pu. A practical and modular method to implement extended transaction models. In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[2] F. Casati, S. Ceri, B. Pernice, and G. Pozzi. Workflow Evolution. In *Proceedings of the 15th International Conference on Conceptual Modeling (ER'96)*, volume 1157 of *Lecture Notes in Computer Science*, Cottbus, Germany, October 1996. Springer-Verlag.

[3] Q. Chen and U. Dayal. A Transactional Nested Process Managemnet System. In *Proceedings of the 1996 IEEE Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

[4] S.-W. Chen. *Recovery for Extended Transaction Models*. PhD thesis, Department of Computer Science, Columbia University, February 1997.

[5] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*, Florida, 1996.

[6] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, 1993. Invited paper.

[7] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of 1990 ACM SIGMOD*, pages 204–214, Atlantic City, New Jersey, June 1990.

[8] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1993.

[9] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, Apr. 1995.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] R. Günthör. The Dependency Manager: A Base Service for Transactional Workflow Management. In *Proceedings of the 6th International Workshop on Research Issues in Data Engineering - Interoperability of Nontraditional Database Systems (RIDE-NDS'96)*, New Orleans, Louisiana, February 1996.

[12] W. Hseush and C. Pu. A Practical Technique for Asynchronous Transaction Processing. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[13] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. `http://www.parc.xerox.com/spl/groups/eca/pubs/complete.html`.

[14] 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, 1996.

[15] L. Liu and C. Pu. A Transactional Activity Model for Organizing Open-ended Cooperative Activities. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS-31)*, Big Island of Hawai'i, HI, January 1998.

[16] L. Liu and C. Pu. Methodical Restructuring of Complex Workflow Activities. In *Proceedings of the 1998 IEEE Conference on Data Engineering*, Orlando, Florida, February 1998.

[17] C. Mohan, G. Alonso, R. Günthör, and M. Kamath. Exotica: A Research Perspective on Workflow Management Systems. *IEEE Data Engineering Bulletin*, 18(1), 1995.

[18] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[19] C. Pu, R. Barga, T. Zhou, and S.-W. Chen. Implementing Extended Transaction Models. In *High Performance Transaction Systems (HPTS) Workshop 1997*, Pacific Grove, California, September 1997.

[20] C. Pu, G. E. Kaiser, and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th International Conference on Very Large Data Bases*, 26-37, August 1988.

[21] C. Pu, T. Autrey, A. Black et al. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Colorado, December 1995.

[22] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a Cooperative Transaction Model: the Cooperative Activity Model. In *Proceedings of the 21st International Conference on Very Large Databases*, pages 194–205, Zurich, Switzerland, September 1995.

[23] D. Shasha, E. Simon, and P. Valduriez. Simple Rational Guidance for Chopping up Transactions. In *Proceedings of 1992 SIGMOD International Conference on Management of Data*, pages 298–307, May 1992.

[24] A. Sheth, D. Georgakopoulos, S. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, and A. Wolf. Report from the NSF Workshop on Workflow and Process Automation in Information Systems. Computer Science Department Technical Report UGA-CS-TR-96-003, University of Georgia, Athens, Georgia, 1996.

[25] H. Wachter and A. Reuter. The ConTract Model. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1993.

[26] D. Wodtke, J. Weissenfels, G. Weikum, and A. K. Dittrich. The Mentor Project: Steps Towards Enterprise-Wide Workflow Management. In *Proceedings of the 1996 IEEE Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

[27] T. Zhou and C. Pu. Recovery Methods for Dynamic Transactional Activities. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1997.

[28] T. Zhou, C. Pu, and L. Liu. Adaptable, Efficient, and Modular Coordination of Distributed Extended Transactions. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, December 1996.

[29] 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, May 1997.

[30] T. Zhou, C. Pu, and L. Liu. Using OCP to Build Flexible Coordination Protocols: A Transactional Multimedia Application. In *Proceedings of International Symposium on Digital Media Information Base (DMIB'97)*, Nara, Japan, November 1997.