UTML: Unified Transaction Modeling Language

Nektarios Gioldasis – Stavros Christodoulakis
Laboratory of Distributed Multimedia Information Systems and Applications
Computer Electronics Department - Technical University of Crete
MUSIC/TUC
{nektarios,stavros}@ced.tuc.gr

Abstract

Web transactions may be complex, composed of several sub-transactions accessing different resources including legacy systems. They may also have complex semantics. To deal with complex web applications, transaction design methodologies and tools need to be very flexible allowing for designing web applications from scratch (top-down design), as well as using existing systems or services to compose new applications which offer added value services (bottom-up design) to the user. In this paper we describe UTML as a high level transaction design language to facilitate the complex web transaction design process. UTML is based on a transaction meta-model, which can describe, in a flexible and extensible manner, most of the known transaction models as well as new ones according to the application's requirements. It provides modeling for transactions that incorporate different behavioral patterns, and it is capable to describe activities with weaker transactional semantics that they do not have all the ACID properties. Unlike other models, it can be used to synthesize new transactions from pre-existing transaction systems (like legacy systems), with diverse transactional semantics. UTML provides a rich notation to visualize the design process using UML class diagrams to model the static structure of transactions and UML state machines to model their dynamic behavior and their flow of execution.

1. Introduction

The first web applications were just information-presentation oriented applications. They provided the user with the ability to navigate from page to page and see information. However, modern web applications become more and more data intensive allowing the user to interact with them and execute complex tasks. They may be composed of several activities accessing many different distributed resources and they may utilize remote services with different semantics and interfaces. Moreover, many web applications utilize existing legacy systems, with diverse semantics, offering to the web user services with combinations of new and existing functionality. The transaction designer in these environments does not have the flexibility to design transactions using a uniform transaction model which structures sub-transactions in a top-down fashion since the sub-transactions are executed by different existing transaction systems which may follow different transaction models and have different transaction semantics. The transaction design is necessarily done in a bottom-up way, and the transaction model should support this process allowing transactions with internal structure and different semantics for the sub-transactions.

A significant difference between centralized applications and web applications is the user behavioral model, assumed by the application designer and supported by the application development tools and the underline infrastructure. Centralized applications usually target a user who knows what he wants and proceeds to complete a focused task, whereas web applications have as model user one that browses around with great flexibility and with a tendency to look around initiating possibly various tasks in parallel without paying much attention to the overheads incurred by open transactions and without paying much attention to close those transactions. Web browsers encourage such flexibility in the user behavior with the browsing flexibility that they provide.

It is possible to place several restrictions in the web user navigational patterns (including the use of the web browser capabilities). However, such restrictions should be as limited as possible to avoid confusing the users of the application and making the application "user-unfriendly". Therefore, web applications and transactions models should provide the users with a great flexibility, allowing the simultaneous opening and closing of several sub-transactions without violating the necessary transaction semantics, and supporting transactions with long life.

The above requirements for web applications show that web transactions can be extremely complex for well designed applications, and that tools that support the web transaction design process are valuable, more valuable than in centralized applications.

Transaction models that provide for transactions with complex internal structure are known as Extended Transaction Models (ETM) and up to now several different such models have been proposed (sagas, nested,

open nested transactions, ConTract, etc) . Each such transaction model provides for transaction modeling from a specific point of view. Others try to come up with the long duration of transaction execution, while others to provide enhanced concurrency releasing locked resources as soon as possible. Some recent web standards have been adopted [OMG 2001a], [Open Group 1992], and new proposals are continuously appearing [OMG 2001b].

Although the ETMs and the adopted standards are valuable in many application domains relaxing some of the ACID transactional properties and providing for distributed transaction management, they can't always deal with the full complexity that some modern web applications have. Their limitations come mainly form their inflexibility to incorporate different transactional semantics as well as different behavioral patterns into the same structured transaction. They are also rigid in describing units of work that have weaker transactional semantics than ACID and are incorporated in the same structured transaction.

In this paper we propose UTML (Unified Transaction Modeling Language) as a general mechanism for transaction modeling. UTML is based on a very flexible and extensible transaction model that can accommodate structured transactions containing sub-transactions with diverse semantics which can be used in a bottom-up design process. The model allows great flexibility for the web user navigational patterns, and also accommodates long-lived transactions. UTML is an extension of UML providing for modeling, documenting and maintaining large scale transactional web information systems.

UTML has been developed in the context of 6th framework IST program of EU, in the project UWA (Ubiquitous Web Applications IST-2000-25131), which is developing and evaluating design methodologies for Ubiquitous Web Applications [www.uwaproject.org].

In section 2 we describe the related work and its limitations while in section 3 we present our objectives for UTML. In section 4 we introduce the fundamental UTML concepts and in section 5 we present the UTML notation. Section 6 shows how UTML can be used to facilitate the transaction design process of web applications, while section 7 presents our conclusions and our future work on UTML.

2. Related Work

Researches in many research areas (databases, transactions, workflows, and web application development) have proposed a number of transaction models that can be used for different application domains. The common characteristic of the more advanced of those models is the internal structure that they propose for transactions. Each one proposes a specific way in which the transaction decomposition is done, according to the point of view that it approaches the problem.

The Nested Transaction model [Moss 1981] for example defines internal structure for transactions in a way that it preserves atomicity and consistency for the top level transaction. On the other hand, the Open Nested Transaction model provides internal structure for transactions relaxing their atomicity by allowing the sub-transactions of a top-level transaction to commit prior to their parent commitment. Both the Nested and the Open Nested transaction model propose a hierarchical tree-like structure, where a top-level transaction is recursively decomposed into sub-transactions.

The SAGAS transaction model [Garcia-Molina and K. Salem 1987] approaches the problem from a different point of view and tries to provide both unlocking of resources prior to the transaction's commitment and hard save points during the execution of the transaction. A SAGA is a sequence of ACID transactions, steps for short, which are executed successively. When a step completes, it commits and makes its results visible to other SAGAS. For each step, a compensation is defined. A compensation is a transaction that is used to semantically undo another transaction. Thus, if a step of a SAGA aborts, the process continues by executing the compensations of successfully executed steps in a reverse order.

The ConTract transaction model [H. Wächter, A. Reuter 1992] allows transactions with internal structure and it also provides an execution script that defines the sequence of transaction execution. However, the definition of the execution sequence is based on a scripting language, which makes the transaction design process difficult since there is no way to design the flow of transaction execution without dealing with any programming language or to define alternative execution flows. Also, each sub-transaction of a ConTract is an ACID transaction and thus the model does not provide for modeling units of work with weaker transactional semantics. Finally, the model does not describe the decomposition semantics of each transaction into sub-transactions (whether its sub-transactions make their results visible to other ConTracts or not).

ACTA was proposed [Chrysanthis and Ramamrithan 1994] as a framework for specifying and reasoning about the structure and behavior of complex transactions. ACTA is not a transaction model itself, rather it is a meta-model capable to describe known transaction models (as well as new ones) and to facilitate their analysis. With this framework it is possible to characterize the whole spectrum of dependencies between

transactions, as well as effects of transactions on accessed objects. It proposes two main transaction dependencies (commit dependency and abort dependency) and it models the effects that transactions have on objects by introducing two object sets. The ViewSet is the set of objects possibly accessible by a transaction and the AccessSet is the set of objects that a transaction has accessed. In general, ACTA provides a rich set of concepts for specifying the transaction behavior and improving concurrency and recovery properties.

The common characteristic of all these models is that they define once the transaction specification (decomposition semantics, commit or abort dependencies between parent and sub transactions and the sequence of messages exchanged between transactions and sub-transactions), and this specification holds for the whole structured transaction model. There is no flexibility in incorporating sub-transactions with different behavior (for example decomposing a transaction into two sub-transactions, one that commits and makes its results visible to others, and one that it keeps its results invisible until the parent commits) and semantics (vital and non-vital sub-transactions) into the same structured transaction. Thus they are appropriate for top-down transaction design, but not for bottom-up transaction design using pre-existing transactional services with diverse semantics. In bottom-up design we need to utilize pre-existing systems or services, which may not have identical behavior (for example utilizing legacy transaction systems that do not support all the ACID properties or do not support the Two Phase Commit Protocol) and the application has to adapt to this pre-existing logic.

None of those models (with the exception of ConTract) provides high level mechanisms for modeling the flow of transaction execution and the constraints on starting the execution of a transaction (modeling the paths that can be taken to start the transaction execution with respect to the application's state). As far as we know, most of these models cannot describe weak transactions (transactions that do not support all the ACID properties) without decomposing them into sub-transactions. The ability of describing weaker flat transactions is very important and would be useful for designing and documenting applications that exhibit transactional behavior, even not so rigid.

The above limitations make clear the need for a transaction design language based on a rich meta-model, which will provide for modeling transactions conforming to known transaction models or new ones, according to the requirements of a specific applications. UTML has been designed to remedy these limitations and accommodate the web application transaction needs. It is UML compatible in order to make use of its well established design mechanisms and conform to a world wide acceptable modeling standard.

3. Objectives

Our objective is to facilitate the complex design process for web transactions by providing a formal, extensible modeling language for web transactions. This language must have the following features:

1. Modeling both the static structure and dynamic behavior of transactions.
2. Modeling transactions compatible to known transaction models.
3. Description of transaction models from scratch (in an extensible manner). As new models may be needed according to the application's requirements the ability to specify new transaction models becomes very important.
4. Modeling different transaction decomposition semantics and behavior into the same structured transaction. With this ability the same transaction can access different resources and utilize legacy systems or services adapting to their behavior.
5. Modeling activities with weaker transactional semantics that they do not have all the ACID properties.
6. Allowing for flexibility in the design so that the web user is not restricted in his behavior by a strict transaction model. The model should allow navigation in and out from sub-transactions and support long-lived transactions.

We propose UTML as an extensible modeling language that can be used from a designer to describe transactional behaviors according to the application's requirements and complexity. UTML is based on a transaction meta-model capable to cope with all the dimensions of extended transactions and it is enhanced with a rich notation to visualize the modeling process.

4. UTML – Unified Transaction Modeling Language

UTML is based on a transaction meta-model. The meta-model includes a number of modeling concepts that are described in this section.

4.1. Operations and Activities

Web applications give the ability to interactive users to invoke certain user-triggered operations. User-triggered operations are interfaces of the application functionality to the interactive user. However, user-

triggered operations are not the only operations of web applications. Other primitive or complex operations may be triggered by the application logic (operations specified to execute in a particular sequence or as a result of the occurrence of some events).

Definition I : An Operation P is a not-suspendable atomic unit of work that is not further decomposed.

The above definition states that an operation has the following characteristics:

- Not-suspendable: An operation cannot be suspended and continue its execution later on.
- Atomic: Either all defined work is executed successfully, or not at all.

By saying that an operation is not further decomposed, we mean that in this meta-model, an operation is the smallest piece of work that can be executed and we are not interested in modeling the internal structure and the logic that implements it.

Operations export system's logic to the end user and may be grouped together to satisfy the achievement of a specific user goal or to satisfy constraints on possible operation invocations and they define execution flows of operations. We use the concept of *Activity* to describe logical parts of functionality that may impose constraints on possible operation invocations.

Definition II : *An **Activity** A is a set of operations and possibly other activities with an optional flow of execution.*

Activities, like operations, have constraints on when they can start, and status which can change when specific Signals appear. A signal is the data that represent the occurrence of some event. A status change may be to start an activity or sub-activity, to abort or suspend an activity and so on.

Activities can also have specific execution contracts with the system. Such contracts define additional semantics and constraints (e.g. all operations executed successfully or not at all) on the activity execution. The different execution contracts that an activity can have are described later in section 4.2.

The complete specification of the activity includes the following fundamental concepts. Here we present only the definition of these concepts without their relationships and mathematical expressions.

- **OperationSet** of an activity A, OS(A), is the set of operations that can be invoked in the scope of this activity. Some of these operations are obligatory while others are optional
- **ManagementSet** of an activity A, MS(A), is the set of operations that belong to its OperationSet and are used to manage the activity. Such operations are: begin, commit, suspend, resume, abort, end, begin_sub, delegate, etc.
- **FunctionalSet** of an activity A, FS(A), is the set of operations that belong to its OperationSet and are used to implement the logic of the activity.
- ***InitializationSet*** *of an activity A, IS(A), is the set of operations that belong to its ManagementSet and are used to start the activity.*
- **TerminationSet** of an activity A, TS(A), is the set of operations that belong to its ManagementSet and are used to terminate the activity.
- ***PropertySet*** *of an Activity A, PS(A), is the set of properties that this activity supports. Such properties are Atomicity, Consistency, Isolation and Durability.*
- ***ActivitySet****, AS(A), of a composite activity A is the set of activities that are sub-activities of A.*

The above concepts are capable to completely specify a simple activity. Some of the previously described sets are extensible (*MS*, *TS* and *IS*) and can be enriched with primitive operations in order to specify different behaviors for activities. These sets constitute the first part of the extensibility mechanism of UTML. The second part is the notion of well-formedness rules, which are used to define formal constraints and dependencies between activities. Well-formedness rules are presented in detail in section 4.4.

Operations and activities that belong to the OperationSet and ActivitySet of an activity respectively can be obligatory or optional. An operation or activity is obligatory when its successful execution is required for the successful execution of the activity in which it is contained.

In complex structures of activities and operations it is needed to keep track of any executed operation and activity. To do so, we define a set of concepts that are used to achieve this goal and represent real time structures and are not used to specify, at design time, the activity. Rather are used to formalize, through the well-formedness rules, the real time behavior of activities, as well as inter-activities dependencies.

- **OperationHistory**, OH(A), of an activity A is the set which contains all the executed so far operations of A, ordered by their time of execution completion.

- **ActivityHistory**, AH(A), of an activity A is the set which contains all the executed sub-activities of A, ordered by the time of their Termination Operation execution.

OperationHistory and ActivityHistory sets can be expressed in a multi-level way, since the activity structure can be hierarchical.

4.2. Execution contracts of Activities

As we mentioned above, activities have a PropertySet, which contains some special properties defining their type. According to the properties that an activity has, its behavior is different and it has to obey a different execution contract. This set is a sub-set of {Atomicity, Consistency, Isolation, Durability}. For example, when the execution contract of the activity includes all these properties, then we have an activity behaving like a traditional ACID transaction. Our meta-model allows more flexible execution contracts that have to be obeyed by activities. However, there are some constraints that have to be taken into account when we define the execution contracts of activities.

The Consistency property refers to the data consistency. That is, any permanent data modification that an activity makes, it must leave data in a consistent state. It is obvious that an activity must support *Durability* in order to support *Consistency*. Otherwise it is meaningless to say that an activity does not modify any data and at the same time to say that this activity leaves data in a consistent state. This constraint can be expressed as follows:

$A : Activity; P, P' : properties$

$$if \exists P \mid P \in PS(A) \wedge P = "Consistency" \quad then \exists P' \mid P' \in PS(A) \wedge P' = "Durability" \qquad \text{Constraint(1)}$$

The type of an activity A is formed using the initials of each property belonging to its PropertySet. For example, the type of an activity that has in its PropertySet the *Atomicity* and *Isolation* properties is *AI_Activity*.

The different activity types that can be defined are:

Activity, A_Activity, I_Activity, D_Activity, AI_Activity, AD_Activity, DI_Activity, DC_Activity, ADI_Activity, ADC_Activity, DCI_Activity, and ACID_Activity.

4.3. Compensations

An activity may have an associated compensation. The compensation may be invoked if the user or the system wants to convert an activity that terminated successfully to an aborted activity.

Definition III : Compensation is a special type of activity that is used to semantically undo a successfully executed activity.

Compensations are activities and thus they are related with all other concepts that activities relate. An exception here is the properties that a compensation can have in its PropertySet. The compensation PropertySet must contain at least the property *"Atomicity"*. This constraint has to be satisfied in order to assert that all the defined operations of a compensation will be executed successfully or not at all and in case of failure it has to be re-attempted.

Given the above constraint, the different compensation types that we have defined are:

A_Compensation, AI_Compensation, AD_Compensation, ACD_Compensation, AID_Compensation, and ACID_Compensation.

4.4. Well-Formedness Rules

Well-formedness rules are formal constraints that define activity dependencies in complex activity models (transaction models). They constitute the second part of the extensibility mechanism of UTML and can express both structural and behavioral constraints of activity models.

The syntax of these rules has two parts. The first part is the documentation of the rule in a natural language (e.g. English), while the second part is a formal mathematic expression of the rule. Currently we have defined 10 such rules, but we present here only one example due to the limited space.

Rule I

- ***Part A)*** A composite activity *A* that supports the *Isolation* property cannot have a sub-activity *B*, which does not support the *Isolation* property. That is, *Isolation is Transitive***.**
- ***Part B)***

$$P, P' : properties ; A, B : activities$$

$$If \exists P = "Isolation" \in PS(A) \Rightarrow \forall B \mid B \in AS(A), \exists P' = "Isolation" \in PS(B)$$

Well-formedness rules typically formalize the modeling process by preventing the designer of misusing the presented concepts in defining complex activity structures. They are also used to specify the co-ordination of activities defining the sequence of messages that have to be exchanged between parent and sub activities during the commitment process. It should be also noted that well-formedness rules can be expressed through OCL (Object Constraint Language) but this requires a specific transaction structure (with a UML class diagram).

4.5. Activity Decomposition

To provide for a great flexibility in activity decomposition, we model the ***decomposition association***, denoted *DA(A,B)*, between a parent, *A*, and sub, *B*, activity. Such an association has two properties.

- ***Vitality*** defines whether the sub-activity is obligatory or optional. A vital sub-activity means that its successful execution is required for the successful execution of the parent activity.
- ***Visibility*** defines whether the sub-activity makes its results visible to any other activity when it completes (prior to the completion of its parent) or not.

Thus, the different types of decomposition associations that can be used are:
Inv_SubActivity, Vis_SubActivity, Vital_Inv_SubActivity, and Vital_Vis_SubActivity.

5. UTML Notation

To visualize the design process, UTML is enhanced with a rich notation that exports the meta-model's functionality in a simple and concrete interface. The notation is based on the Unified Modeling Language and actually consists of a UML profile (a set of UML stereotypes appropriate for modeling specific applications). To simplify the design process we propose many different stereotypes, rather than having "heavy" stereotypes with many properties that will have to be set by the designer.

The defined profile consists of two different UML models: the Organization Model and the Execution Model. The Organization Model represents the designed activities from a static point of view, while the Execution Model describes the execution flow of activities and their real time dependencies. For each model, a set of UML stereotypes has been defined. For each stereotype there is a complete specification as defined by the UML specification [OMG 2000].

5.1. Organization Model Stereotypes

The Organization Model of UTML captures the system from a static point of view. It describes the complete specification and the structural dependencies of activities. Also, it models the decomposition of activities into sub-activities in terms of visibility and vitality. The notation that we use for Organization modeling of activities is based on UML class diagrams. In each diagram appropriate stereotyped classes are used. A stereotyped class represents an activity (conceptually) of type that the stereotype indicates. Each stereotype has been completely defined as UML 1.4 [OMG 2000] specification specifies. Here, we will show the complete specification of only one stereotype. The same procedure has been followed for every defined stereotype.

Stereotype	**Atomic Activity** «A_Activity»
Base Class	Class
Parent	NA
Description	…
Constraints	
Tags	Name String TriggeredBy String <user, appLogic> isSimple Boolean isStrict Boolean isSynchronous Boolean TimeOut Integer mSet String <management operations> iSet String <Initialization operations> tSet String <Termination operations> Documentation Text
Notation	A class shape stereotyped as «Activity»

Using the same specification pattern, the following stereotypes have been defined for the Organization Model:

Stereotype	Base Class	Stereotype	Base Class
«OrganizationModel»	Model	«ACID_Activity»	Class
«OrganizationPackage»	Package	«Compensation»	Class
«Activity»	Class	«AD_Compensation»	Class
«A_Activity»	Class	«AI_Compensation»	Class
«I_Activity»	Class	«AID_Compensation»	Class
«D_Activity»	Class	«ACD_Compensation»	Class
«AI_Activity»	Class	«ACID_Compensation»	Class
«AD_Activity»	Class	«Requires»	Association
«DI_Activity»	Class	«Inv_SubActivity»	Association
«DC_Activity»	Class	«Vis_SubActivity»	Association
«ADI_Activity»	Class	«Vital_Vis_SubActivity»	Association
«ACD_Activity»	Class	«Vital_Inv_SubActivity»	Association
«DIC_Activity»	Class	«Compensates»	Association

5.2. Execution Model Stereotypes

For the Execution Model the following stereotypes have been defined:

Stereotype	Base Class	Stereotype	Base Class
«ExecutionModel»	Model	«Commit»	PseudoState kind
«ExecutionPackage»	Package	«Rollback»	PseudoState kind
«ExplicitStart»	State		

The stereotypes «ExplicitStart», «Commit» and «Rollback» are of PseudostateKind type and each one represents a specific state during which the application it does not execute a specified activity to achieve a particular user goal. While the use of «Commit» and «Rollback» stereotypes is obvious, the «ExplicitStart» stereotype should be further explained. It is used inside optional activities, which are represented as states in the execution model, to denote that the user explicitly mast start them. When optional activities are represented as parallel sub-states (called regions), their activation is done by default since all regions of a state are entered simultaneously. To overcome this situation, we have defined the «ExplicitStart» pseudostate, which is used to declare that the user, explicitly mast start or terminate the optional activity.

6. UTML in Practice

Currently, UTML is capable to describe transactions conforming to the most known transaction models (nested, open nested, sagas, ConTracts, etc.). In sections 6.1 and 6.2 we will provide two examples on how this can be achieved. The same procedure can be followed for describing transactions conforming to other known transaction models. In section 6.3 we provide an extended example with custom transactions that are enough flexible and incorporate different behavioral patterns. Also, the example used there shows how UTML facilitates a bottom-up design process.

6.1. Describing Nested Transactions with UTML

In the Nested Transaction model, a top-level transaction is recursively decomposed into several sub-transactions. When a sub-transaction completes it does not make its results visible to other nested transaction, but it passes its results to its parent. A parent transaction completes when all its sub-transactions have completed. In UTML a transaction conforming to this model can be described as follows:

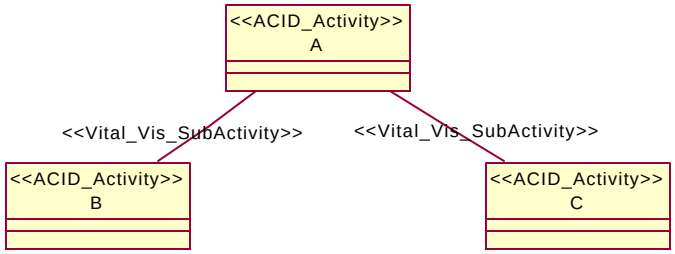

Figure 6.1.1 A Nested Transaction

The following table shows the main specification properties (as they have been defined in the corresponding stereotypes) of each one of the activities A, B, and C.

A	B	C
isSimple= *False* mSet =*suspend, resume* iSet=*begin* tSet= *commit, abort*	isSimple=*True* mSet=*suspend, resume* iSet= *begin_vital_sub* tSet=*delegate, abort*	isSimple=*True* mSet=*suspend, resume* iSet= *begin_vital_sub* tSet=*delegate, abort*

Of course the whole specification of the above activities includes many other properties. However, the aforementioned ones are enough to show that the described transaction conforms to the nested transaction model. To completely specify the transaction behavior, we need to provide the execution model of the transaction and its well-formedness rules. Since the execution model for the above example is very simple, we will show only the well formedness rules that are required to specify the behaviour of the transaction.

Rule 6.1.1:

Description: In order for the activity A to commit, the sub-activities B and C must have completed successfully (by delegation).

Expression:

$$O, O', O'' : operations; A, B : activities$$

$$\exists O \in OH(A) \mid O = "Commit" \Rightarrow \exists O' = "delegate" \in OH(B) \wedge \exists O'' = "delegate" \in OH(C)$$

Rule 6.1.2:

Description: Activities B and C must start only after the activity A has started.

Expression:

$$O, O', O'' : operations; A, B, C : activities$$

$$\exists O = "begin\_vital\_sub" \in OH(B) \vee \exists O' = "begin\_vital\_sub" \in OH(C) \Rightarrow \exists O'' = "begin" \in OH(A)$$

6.2. Describing Open Nested Transactions with UTML

In the Open Nested Transaction model, a top-level transaction is recursively decomposed in several sub-transactions. When a sub-transaction completes it makes its results visible to other transactions. A parent transaction completes when all its sub-transactions have completed. If some sub-transaction fails, its parent has to abort and this abortion may includes compensation of other successfully completed sub-transactions. In UTML a transaction conforming to this model can be described as follows:

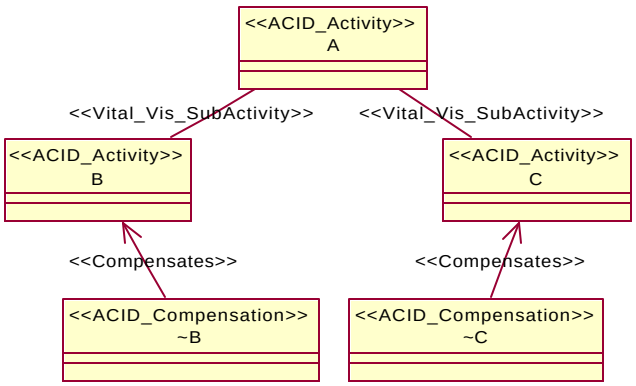

Figure 6.2.1 An Open Nested Transaction

The specification of the activities involved in the above open nested transaction, is presented below:

A	B	C
isSimple= *False* mSet =*suspend, resume, compensate* iSet=*begin* tSet= *commit, abort*	isSimple=*True* mSet=*suspend, resume* iSet= *begin_vita_visl_sub* tSet=*commit, abort*	isSimple=*True* mSet=*suspend, resume* iSet= *begin_vital_vis_sub* tSet=*commit, abort*

For the compensations that are used in the example, its specification has as follow:

~B	**~C**
mSet=none iSet=begin tSet=commit, abort	mSet=none iSet=begin tSet=commit, abort

The following well-formedness rules have been defined for the above example:

Rule 6.2.1:

Description: In order for the activity A to commit, the sub-activities B and C must have previously committed.

Expression:

$$O, O', O'' : operations; A, B, C : activities$$

$$\exists O \in OH(A) \mid O = "Commit" \Rightarrow \exists O' = "commit" \in OH(B) \wedge \exists O'' = "commit" \in OH(C)$$

Rule 6.2.2:

Description: Activities B and C must start only after the activity A has started.

Expression:

$$O, O', O'' : operations; A, B, C : activities$$

$$\exists O = "begin\_vital\_vis\_sub" \in OH(B) \vee \exists O' = "begin\_vital\_vis\_sub" \in OH(C) \Rightarrow \exists O'' = "begin" \in OH(A)$$

Rule 6.2.3:

Description: The rolling back of the activity A includes the compensation of any succes sfully executed sub-activity of A.

Expression:

$$O : operation; A, A_i : activities$$

$$\exists O = "abort" \in OH(A) \Rightarrow \forall A_i \in AH(A) \exists \sim A_i \in AH(A)$$

6.3. The Conference Manager System Example

To further exemplify the use of UTML we provide an extended example of transactional web application. The Conference Manager System is an application used by the attendees of a conference. This application provides complete functionality for paper submission, paper refereeing, conference registration, conference trip planning, etc. However, in this example we show only one activity, which is enough complicated and describes some aspects (even not all) of the problem. This activity is the Plan Conference Trip (PCT) activity, which includes User Authorization, Hotel Reservation, Airline Tickets Reservation and Social Event Tickets Reservation (ticket reservation for some social events during the conference days).

The application accesses distributed databases (databases of the airline businesses, hotels and the organizations offering the social events). Also, it utilizes a web service offering some critic about the social events. This service is pay-per-use and since the user calls it, he has to pay for, independently to whether he will make a reservation or not. The payment for ticket reservation is done through a bank. This means that the application has to utilize the functionality (Debit Amount-DA) of the bank's legacy system.

Hotel Reservation (HR) and Airline Ticket Reservation (ATR) are both vital activities, while Social Event Tickets Reservation (ETR) is an optional activity. That is, the user may or may not execute ETR, according to his will. The aforementioned dependencies mean that in order for PCT to terminate successfully both HR and ATR must terminate successfully.

Each one of these three activities is composed of several sub activities. In particular:

- User Authorization:
  - Supply User Info (SUI)
  - Check User Info (CUI)
- Hotel Reservation:
  - Find Hotel (FH)
  - Select Room (SR):
  - Supply Billing Info (SBI)
  - Debit Amount (DA)
- Airline Ticket Reservation:
  - Find Flight (FF)
  - Select Ticket (ST)

- Supply Billing Info (SBI)
- Debit Amount (DA)

- Event Ticket Reservation:
  - Find Interesting Events (FIE)
  - Call Critic Service (CS)
  - Select Tickets (ST)
  - Supply Billing Info (SBI)
  - Debit Amount (DA)

In this example the Plan Conference Trip activity accommodates a number of sub-activities of diverse semantics and behavior. The activities that compose the PCT activity have different transactional semantics. User authorization is an atomic activity while Hotel Reservation is an atomic and durable activity. Actually, User Authorization is not a traditional transaction. On the other hand, Hotel Reservation (HR) is a structured activity which makes data modifications and needs to be atomic and durable. The same holds for ATR and ETR.

More complex is the situation with the internal structure and decomposition of HR, ATR and ETR. Each one of these sub-activities is decomposed into several sub-activities with different decomposition semantics (with terms of vitality and visibility) and no all sub-activities have to be compensated in case of failure. In particular, the Debit Amount (DA) activity utilizes functionality offered by the pre-existing bank's information system and is a traditional transaction with all the ACID properties. In case of failure, an executed instance of DA has to be compensated by the application. Thus an appropriate compensating activity has to be provided. On the other hand, the Critic Service (CS) activity in the Event Ticket Reservation (ETR) activity must not be compensated in case of failure, because it is a pay-per-use service offered by another organization (through its pre-existing system).

We will use UTML to model the static structure and the dynamic behaviour of activities that cooperate to accomplish the user goal of planning the conference trip. Due to limited space we do not provide the full specification of each activity, but only the UTML diagrams that show the language's ability to describe complex activity organizations facilitating the design process. We show the type of each activity, its decomposition (with decomposition semantics), and its internal execution flow.

6.4. Organization Model Diagrams

The following diagrams describe the static structure of the Plan Conference Trip top-level activity:

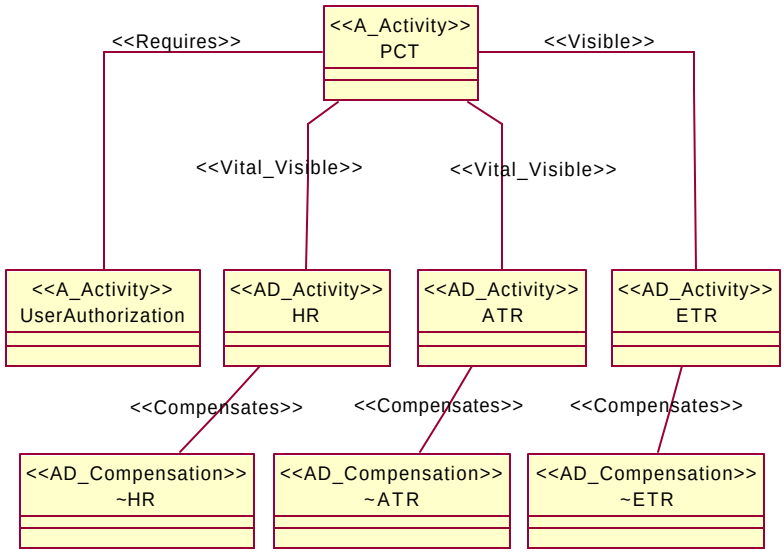

Fig. 1. Plan Conference Trip decomposition

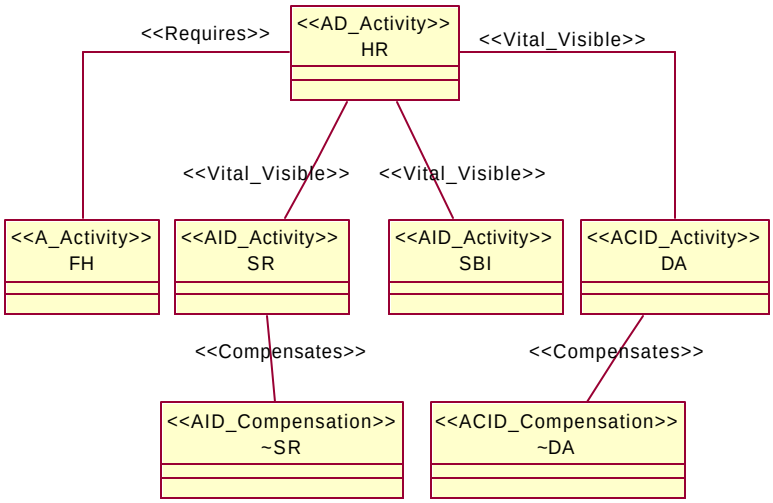

Fig. 2. Hotel Reservation decomposition

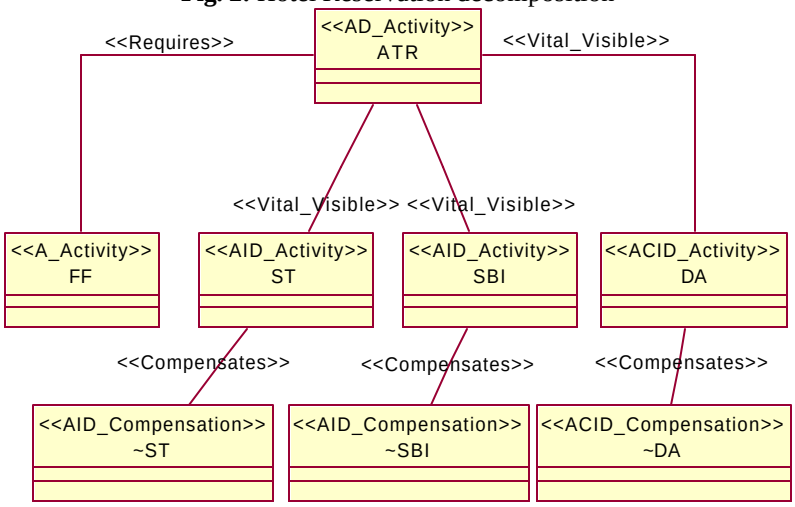

Fig. 3. Airline Ticket Reservation decomposition

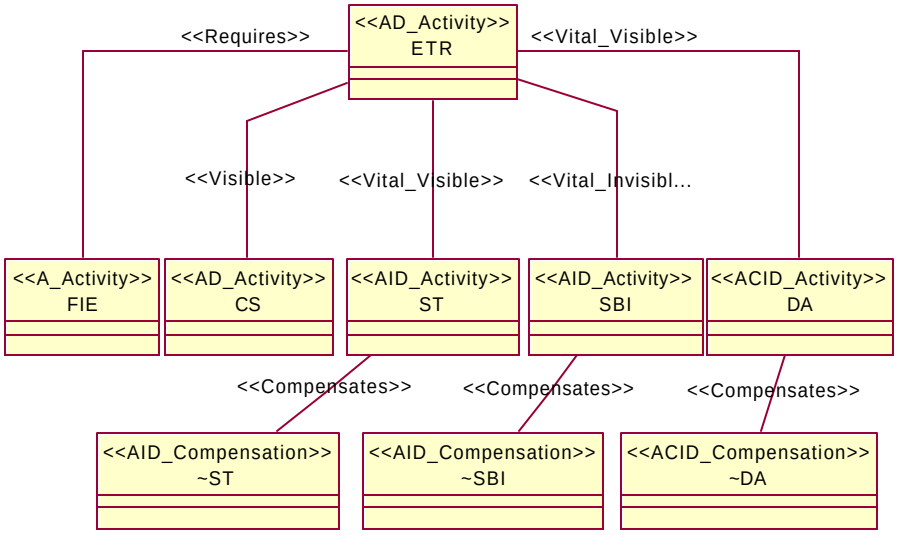

Fig. 4. Event Ticket Reservation decomposition

6.5. Execution Model Diagrams

The following diagrams describe the execution flow for the Plan Conference Trip top-level activity:

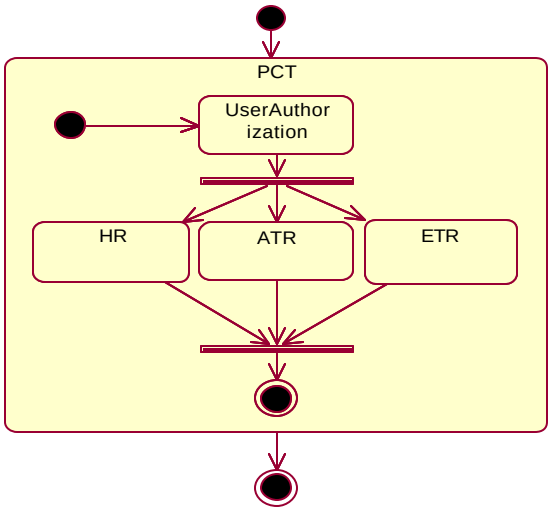

Fig. 5. Plan Conference Trip – Execution Model

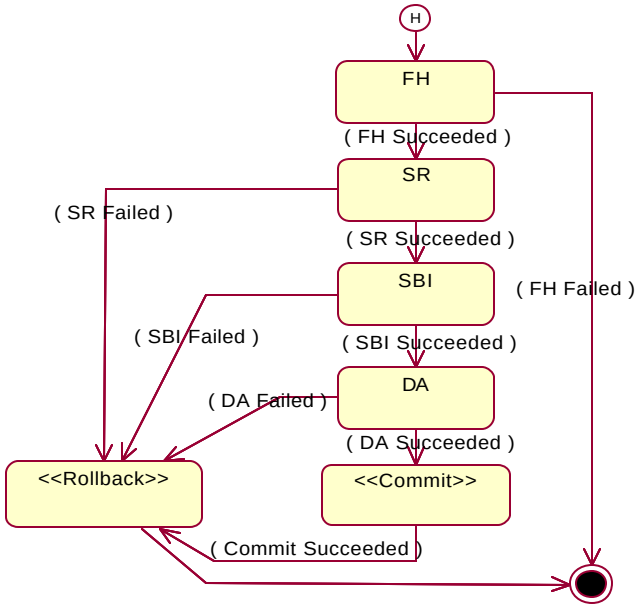

Fig. 6. Hotel Reservation – Execution Model

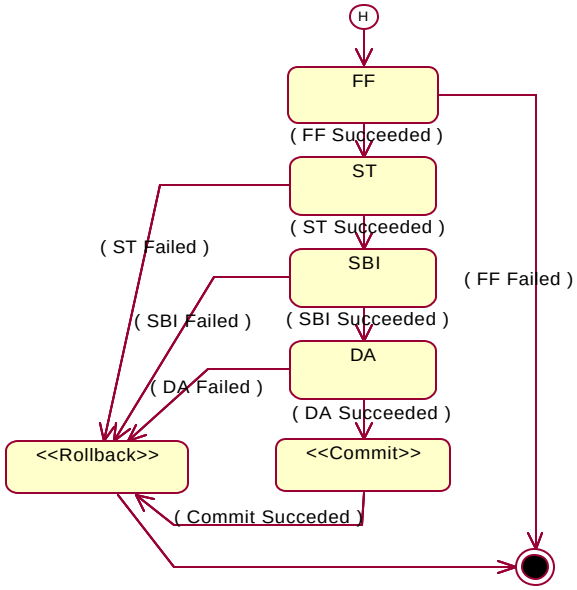

Fig. 7. Airline Ticket Reservation – Execution Model

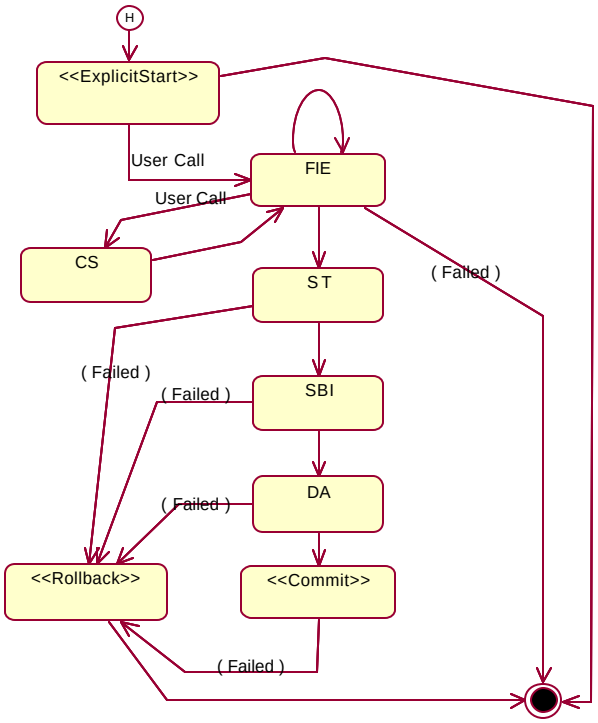

Fig. 5. Event Ticket Reservation –Execution Model

In this example we presented the flexibility that UTML provides in designing complex transactions. We modeled transactions that are decomposed into sub-transactions with diverse semantics and behavior. We also described both the static structure of transactions and their dynamic behavior.

7. Conclusions and Future Work

Transaction design in the web is a very complex task that has to take into account diverse pre-existing resources that are used to synthesize new added value services, as well as the flexibility and the navigational behavior of the web users.

To support the web application design process we need a very general, flexible and extensible transaction model that can accommodate existing transaction models and can support structured diverse sub-transactions and a bottom-up transaction design process. The model should give as less restrictness as possible to the web users while preserving transaction correctness. It should also accommodate long-lived transactions. In this paper we have described such a formal transaction model and the support for designing transactions in this model through UML extensions which we have designed.

This work has been done into the context of the EU project UWA (IST-2000-25131), which aims in the development and evaluation of new tools for the design of ubiquitous web applications. An implementation of the tool for transaction design is currently under development. This tool will provide for modeling transactions with UTML and will export XML documentation for the transaction design. It is expected to be integrated with a set of other tools (for requirements, customization, and hypermedia design) that the UWA partners will provide before the summer of 2002. These tools will be completely evaluated against bank and e-commerce applications.

Also, we are working on the core meta-model on which UTML is based in order to extend it making it capable to describe data flow dependencies between activities and formalize the compensation strategies which can be defined for activities.

Bibliography

[Alonso 1997] G. Alonso: "Processes + Transactions = Distributed Applications". In: Proceedings of High Performance Transaction Processing Systems Workshop 1997. (Also in MiddlewareSpectra, vol.11, no.4), Asilomar, California, USA, September 1997.

[Barbará and Garcia-Molina 1994] D. Barbará and H. Garcia-Molina, "The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems", VLDB J., Vol 2, No 3, 1994.

[Bernstein et al 1987] P. Bemstein A. Hadzilacos and Goodman N. Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, M.A. 1987

[Birilis et al 1994] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish "ASSET A System for Supporting Extended Transactions" Proc. of the ACM SIGMOD International Conference on Management of Data, 1994

[Booch 1994] G. Booch: "Object-Oriented Analysis And Design with Applications, 2nd Edition" 1994

[Bukhres et al 1993] O. Bukhres, A. Elmagarmid, and E. Kuhn: "Implementation of the Flex Transaction Model", Bulletin of the IEEE Technical Committee on Data Engineering, June 1993

[Chrysanthis and Ramamritham 1990] P.Chrysanthis and K. Ramamritham: "ACTA: A framework about Specifying and Reasoning about Transaction Structure and Behavior". In proceed. Of the ACM SIGMOD Int. Conf. on Management of Data, pages 194 – 203, Atlantic City, NJ, May 1990

[Chrysanthis and Ramamritham 1994] P. Chrysanthis and K. Ramamritham: "Synthesis of Extended Transaction Models using ACTA" ACM TODS1994

[Cichocki et al 1998] A. Cichocki, A. Helal, M. Rusinkiewicz and D. Woelk "Wrorkflow and Process Automation: Concepts and Technology" 1998

[Cockburn 2001] Alistair Cockburn: "Writing Effective Use Cases" 2001

[Conallen 1999] J. Conallen: "UML Extension for Web Applications 0.91" http://www.conallen.com

[Dayal et al 1990] U. Dayal, M Hsu, and R. Ladin: "Organizing Long-Running Activities with Triggers and Transactions" In proc. Of the ACM SIGMOD Int'l Conf. On Management of Data, May 1990

[Fowler and Scott 1999] M. Fowler & K. Scott: "UML Distilled: Applying the standard Object Modeling Language" 1999

[Garcia-Molina and K. Salem 1987] H. Garcia-Molina and Kenneth Salem: "SAGAS" In proc. Of the ACM SIGMOD Int'l Conf. On Management of Data, May 1987

[H. Wächter, A. Reuter 1992] "The ConTract Model" Transaction Models for Advanced Applications. Morgan Kaufmann Publishers

[Hasse and Schek 1996] H. Hasse, H.-J. Schek: "Unified Theory for Classical and Advanced Transaction Models". In: Dagstuhl-Seminar "Object-Orientation with Parallelism and Persistance", 1996.

[Jacobson et al 1998] Ivar Jacobson, Grady Booch and James Rumbaugh: "The Unified Software Development Process" 1998

[Korth et al 1990] H. Korth, E. Levy, and A. Silberschatz: "Compensating Transactions: A New Recovery Paradigm" In proc. Of the 16th Int'l conf. On VLDB 1990

[Moss 1981] Moss J. E. B.: "Nested Transactions: An approach to reliable distributed computing" PhD

thesis, MIT, 1981

[O'Neil 1986] P.E. O'Neil, "The Escrow Transactional Model", ACM TODS, Vol 11, No. 4, PP. 405-430, Dec. 1986

[OMG 2000] Object Management Group: "Unified Modeling Language specification"

[OMG 2001a] Object Management Group: "Object Transaction Service Specification"

[OMG 2001b] Object Management Group: "Activity Service Specification"

[Open Group 1992] Open Group: "Distributed Transaction Processing: The XA Specification", X/Open Document C193, ISBN 1-85912-057-1.

[Pu et al 1988] C. Pu, G. Kaiser and N. Hutchinson: "Split Transactions for Open-Ended activities" In proc. Of the 14th Int'l conf. On VLDB 1998

[Rumbaugh et al 1999] James Rumgaugh, Ivar Jacobson and Grady Booch: "The Unified Modeling Language

[Schaad et al 1995] W. Schaad, H.-J. Schek, G. Weikum: "Implementation and Performance of Multi-level Transaction Management in Multidatabase Environment". In: Proc. of the 5th Int. Workshop on Research Issues on Data Engineering: Distributed Object Management, RIDE-DOM'95, Taipei, Taiwan, March 1995.

[Schuldt et al 1999] H. Schuldt, G. Alonso, H.-J. Schek: "Concurrency Control and Recovery in Transactional Process Management". Proc. of the ACM Symposium on Principles of Database Systems (PODS'99), Philadelphia, Pennsylvania, USA, May 31 - June 2 1999

[Sun Microsystems 1999] Sun Microsystems: "JTA: Java Transaction API"

[Terry et al 1994] D. Terry, A.J. Demers, K. Petersen, M.M. Spreitzer, M.M. Theimer, and B.B. Welch, "Session Guarantees for Weakly Consistent Replicated Data" Proc. Conf. P. A.D.C., Austin, Texas, Oct. 1994

[Weikum 1991] G. Weikum: "Principles and Realization Strategies of Multi-Level Transaction Management" ACM TODS, 1991

[Weikum et al 1990] G. Weikum, C. Hasse, P. Broessler and P. Muth: 'Multi-Level Recovery" Proc. Of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems 1990

Subrahmanyam 1999] A. Subrahmanyam: "Nuts & Bolts of Transaction Processing" May 1999