UTML: Unified Transaction Modeling Language

Nektarios Gioldasis – Stavros Christodoulakis
Laboratory of Distributed Information Systems & Applications
Technical University of Crete

{nektarios,stavros}@ced.tuc.gr

Abstract

We propose UTML as a high level transaction modeling language to facilitate the complex web transaction design process. Web transactions may be complex, composed of several sub-transactions and they may access resources with diverse behavior and interfaces like legacy systems and databases. They may also have complex semantics. Thus, 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. 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 transaction design process. This notation has been built on top of UML using its extension mechanisms.

1. Introduction

While the first web applications were just information-presentation oriented applications, which provided the user with the ability to navigate and see information, modern web applications become more and more data intensive applications with a high degree of complexity. Web applications often impose complex transaction processing requirements and there is a need for appropriate tools for designing, documenting, and maintaining this transaction logic.

Web applications may be composed of several activities accessing many different distributed resources and they may utilize remote services with different semantics and interfaces. Many web applications utilize existing legacy systems, with diverse semantics, offering to the web user functionality which is a combination of new and pre-existing logic. In such a case, transactions have to take into account the interface, the semantics and the limitations of pre-existing logic and legacy systems, following a bottom-up design approach. Other transactional web applications may be designed from scratch by decomposing user goals into sub-goals that exhibit transactional behavior or splitting long-lived activities into sub-activities for performance reasons following a top-down design approach.

A significant difference between centralized applications and web applications is the user behavioral model which is assumed by the application designer and is 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 who 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 user navigational patterns (including restrictions on the use of the web browser capabilities). However, such restrictions should be as limited as possible to avoid user confusion and to make the application "user-unfriendly". After all, other competing web applications are just one click away. Therefore, web applications and transaction 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.

Finally, web applications may be presented to users in diverse devices (mobiles, palmtops, etc.) making deviations of the interface and the flow of logic, thus forming families of applications that utilize the same application logic with rather small deviations. Thus, the transactional logic and the precise semantics of such

applications should be well modeled and documented in order to be reusable.

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

Whereas high level modeling methodologies and tools for software and application-logic design have been widely accepted and standardized [19], there are no such mechanisms to facilitate the modeling and design of application logic that exhibits complex transactional behavior.

In this paper we bring together high level modeling mechanisms and transactional aspects of application logic. We propose UTML (Unified Transaction Modeling Language) as a high level transaction modeling language to facilitate the complex transaction design process. UTML is based on a very flexible and extensible transaction meta-model, capable to accommodate structured transactions containing sub-transactions with diverse semantics, and can be used in both top-down and bottom-up design processes. 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 core 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 several 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.

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. Some try to deal with the long duration of transaction execution, while others to provide enhanced concurrency, releasing locked resources as soon as possible. Recently, related web standards have been adopted [18], [20], and new proposals are continuously appearing [17].

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 and 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 into the same structured transaction.

The Nested Transaction model [15] 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's 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 [8] 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, when a step of a SAGA aborts, the process continues by executing the compensations of all successfully executed steps in a reverse order.

The ConTract transaction model [11] 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 or to define alternative execution flows without dealing with a programming language. In addition, 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 ([22], [23]) was proposed 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 dependencies between transactions, and the 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. However, ACTA uses fist-order logic to describe transactions and thus is inappropriate for design and documentation. Also, it lacks the ability of arbitrarily combining diverse transaction models under the same structured transaction.

All these models define 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) once, 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 keeps its results invisible until the parent commits) and semantics (vital and non-vital sub-transactions) into the same structured transaction. Thus, they are inappropriate for bottom-up transaction design using pre-existing transactional systems and 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).

None of those models (with the exception of ConTract) provides 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). No one of the transaction models or meta-models that we know provides for modeling weaker transactions that do not have all the ACID properties. Moreover, these models do not provide a high level design mechanism represented by a familiar notation that can be easily used by application designers. This is very important and would be useful for designing and documenting applications that exhibit transactional behavior.

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 application. 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 powerful design mechanisms and to conform to a well-established industrial modeling standard.

3. Objectives of UTML

Our overall objective is to provide a high level modeling language for applications that exhibit transactional behavior. The produced models of such a language could also be used to document the application's logic and semantics. Note that documentation of the precise transactional semantics of static and dynamic behavior of the application is very important not only to maintain the application, but also to derive many different applications of the same application family (like an application re-implemented or transformed several times for different terminal devices). We have chosen to extend the Unified Modeling Language (UML), in a formal and extensible manner, making it able to describe activities which have complex transactional semantics. UML is a world-wide industry standard for modeling, and UTML has been built on top of it using its extensibility mechanisms. That is, UTML is completely compatible with UML. The objectives that we have set for UTML are:

1) Give to the designer the ability to analyze, design, and describe both the static structure of transactions and their dynamic behavior. We strongly believe that it is important to model not only the structural dependencies of transactions, but also their dynamic behavior and their real time execution dependencies (for example, the flow of their execution, sequential or parallel).
2) Provide appropriate tools for designing transactions compatible to most of the known transaction models, as well as new ones according to the application's requirements.
3) Provide the ability for describing different transaction decomposition semantics and behavior into the same structured transaction. This is very important for applications that access resources (databases, legacy systems, file systems, etc.) with different interfaces, behavior and semantic. With this modeling ability the same transaction can access different resources and utilize existing (legacy) systems or services allowing

more flexibility in the transaction execution without violating the transactional semantics.

4) Support the design of transactions that allow typical user behavior in the web, where users can navigate in and out of transactional activities and they are not necessarily bound to one service provider (other providers are only a click away and over-restricting the user interface may result into loosing customers).
5) Support for designing long-lived transactions, asynchronous transactions, and families with roughly the same application logic but delivering it through different devices (mobile phones, palmtops, etc.)

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 – Meta-model

UTML is based on a rich transaction meta-model. This section describes this meta-model. Figure 1 shows the UTML meta-model as a UML class diagram.

A. 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 be executed in a particular sequence or as a result of the occurrence of some events).

Definition 1: *An operation P is a non-suspendable, atomic unit of work that is not further decomposed.*

The above definition implies that operations have two main characteristics: A) they cannot be suspended and continue their execution later on, and B) either all defined work is executed successfully or not at all. By saying that an operation is not further decomposed we mean that in UTML an operation is the minimum piece of work that can be modeled.

Operations export the application 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 according to a specific execution flow. We use the concept of *Activity* to describe a set of operations implementing logical parts of functionality that

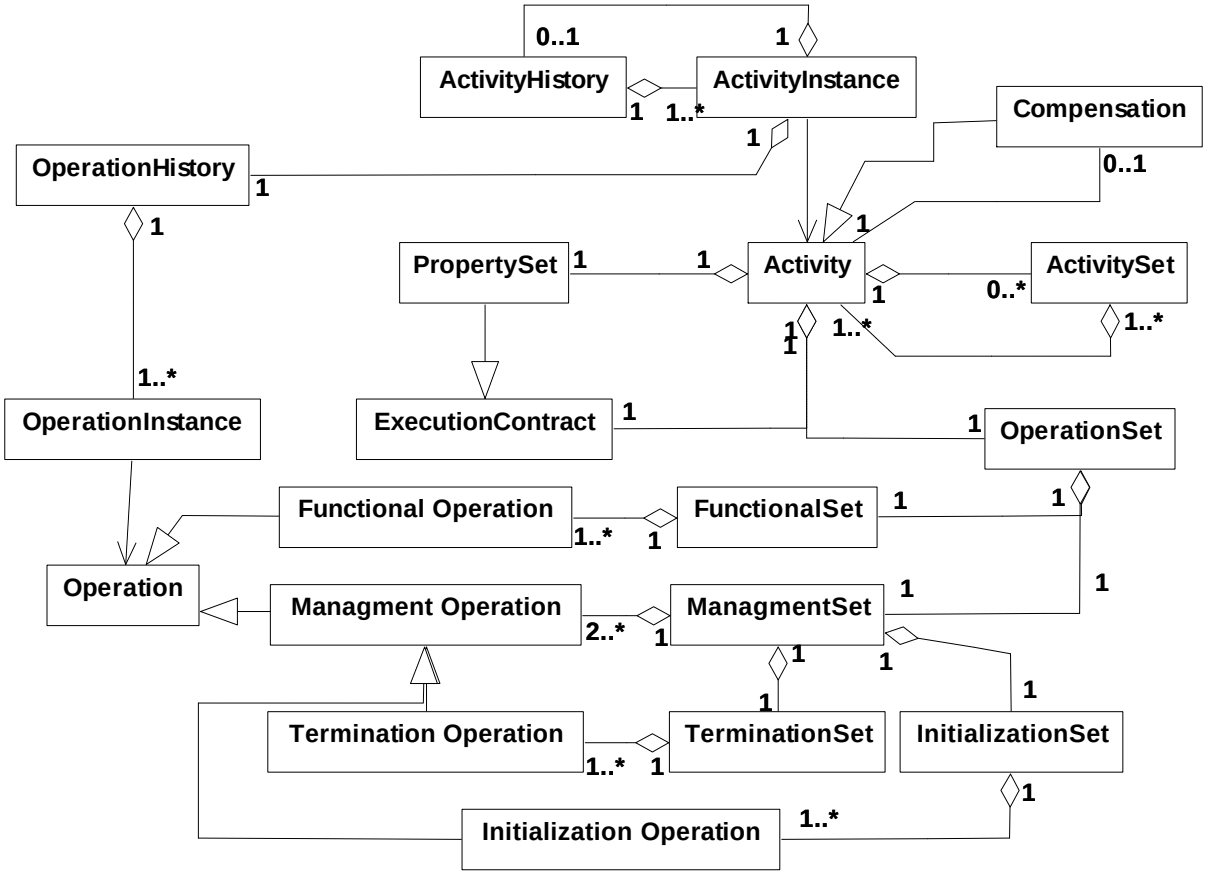

Figure 1: The UTML Meta-model

may impose constraints on possible operation invocations.

Definition 2: *An activity A is a set of operations and possibly other activities with an optional flow of execution defined for them.*

Activities, like operations, have constraints on when they can start, and status which can change when specific *signals* appear. A signal is the occurrence (represented by some data) 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 underlying 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 part B of this section.

Definition 3: *The 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 and some are optional. Also, an important distinction has to do with the operations that are used to implement the activity's logic and those that are used to manage the activity.

Definition 4: *The ManagementSet of an Activity A, MS(A), is the set of operations that belong to its OperationSet and are used to manage the execution of the activity.*

Definition 5: *The FunctionalSet of an Activity A, FS(A), is the set of operations that belong to its OperationSet and implement the activity's logic.*

Management operations are further distinguished into operations that initiate and terminate an activity.

Definition 6: *The InitializationSet of an Activity A, IS(A), is the set of operations that belong to its ManagementSet and are used to initialize the activity.*

Definition 7: *The TerminationSet of an Activity A, TS(A), is the set of operations that belong to its ManagementSet and are used to terminate the activity.*

As we have mentioned, activities may have to obey specific execution contracts with the underline system. Execution contracts are formal sub-sets of the ACID properties and in UTML they are realized by the PropertySet.

Definition 8: *The PropertySet of an Activity A, PS(A), is the set of transactional properties that this activity supports.*

Activities may be associated with compensations. A compensation is a specific type of activity that is used to convert a successfully executed activity to an aborted one. A compensation for example could be used for the cancellation of a hotel reservation.

Unlike operations, activities can be decomposed into sub-activities providing for internal structure. The semantics attached in a decomposition association are expressed in terms of vitality and visibility. A sub-activity is a vital one if its successful execution is required for the termination of its parent activity. Visibility refers to the time that the effects of a sub-activity become visible to other parallel activities. A visible sub-activity commits and makes its results visible to others prior to its parent commitment, while an invisible activity makes its effects visible to others at time of their parent commitment. Vitality and visibility properties of a decomposition association between two activities A and B are denoted as: (AB).vitality=True/False and (AB).visibility=True/False

Definition 9: *The ActivitySet of an Activity A, AS(A), is the set all sub-activities that can be invoked in the scope of this activity.*

The above concepts can adequately support the structural dependencies of transactions. In addition, UTML provides appropriate concepts that can be used to formalize the real time behavior of activity execution. These concepts facilitate in keeping track of any executed operation or activity and they represent real time structures. That is, they are not used to specify the activity at design time, but they are the basis on which appropriate well-formedness rules are expressed.

Definition 10: *OperationHistory of an Activity A, OH(A), is the set of all so far successfully executed operations of A, ordered by the time of their execution completion.*

Definition 11: *The ActivityHistory of an Activity A, AH(A), is the set of all so far successfully 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.

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 (custom transaction models). 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 D.

B. Execution contracts of Activities

As we mentioned above, activities have a PropertySet, which contains some special properties defining their type. The set of properties that an activity has determines its behavior. 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, the process of defining weaker execution contracts is formalized by appropriate well-formedness rules.

Rule 1: *Activities that support the property "Consistency" must also support the property "Durability"*

Rule 2: *Activities that support the property "Consistency" must also support the property "Atomicity"*

The idea behind rule 1 is that the consistency property implies that 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. On the other hand, rule 2 says that since consistency is checked at the termination of the activity it must be ensured that, when detected, inconsistent instances of activities are rolled back. This is provided by defining the activity as atomic.

The type of an activity 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*. Thus, the different activity types that we have defined are: *Activity, A_Activity, I_Activity, D_Activity, AI_Activity, AD_Activity, DI_Activity, ADI_Activity, ADC_Activity and ACID_Activity.*

C. Compensations

As we have mentioned, an activity a may have an associated compensation denoted as c(a). The compensation may be invoked if the user or the system wants to convert an activity that terminated successfully to an aborted 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's 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. Given the above constraint, the possible compensation types defined in UTML are: *A_Compensation, AI_Compensation, AD_Compensation, ACD_Compensation, AID_Compensation and ACID_Compensation.*

D. Well-formedness rules

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

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 represented in UML class diagram. Here we present the rules that are used to formulate the structuring of activities:

Rule 3: *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 downwards transitive.*

$$P, P' \quad properties \; ; A, B \quad activities$$
$$if \exists P \in PS(A) \wedge P = "Isolation" \Rightarrow \forall B \in AS(A) \exists P'$$
$$such \quad that \; P' \in PS(B) \wedge P' = "Isolation"$$

Rule 4: *A composite activity A that has in its ActivitySet a at least one sub-activity that supports the property Durability, must also support durability. That is, Durability is upwards transitive.*

$$P, P' \quad properties \; ; A, B \quad activities$$
$$\forall B \in AS(A) \quad \exists P \in PS(B) \wedge P = "Durability" \Rightarrow \exists P' \in PS(A)$$
$$such \quad that \; P' = "Durability"$$

Rule 5: *If a composite activity A is consistent, then all its visible sub-activities must be also consistent. That is, Consistency is downwards transitive to visible sub-activities.*

$$P, P' \quad properties \; ; A, B \quad activities$$
$$\exists P = "Consistency" \in PS(A) \Rightarrow \forall B \in AS(A) \,|\, (A, B).visibility = true$$
$$\exists P' \in PS(B) \; such \quad that \; P' = "Consistency"$$

Rule 6: *An invisible sub-activity B cannot be further decomposed into visible sub-activities.*

$$A, B, I \quad activities$$
$$(A, B).visibility = false \Rightarrow \forall I \in AS(B) \qquad (B, I).visibility = false$$

Rule 7: *If an activity A has an associated compensation C(A), then C(A) must contain all properties supported by A plus the Atomicity property.*

$$A \quad activity$$
$$PS(C(A)) = PS(A) \cup \{Atomicity\}$$

Rule 8: *Composite activities without any functional operation or at least one durable sub-activity cannot be durable.*

$$A, B \quad activities \; ; O \quad operation \; P, P' \quad properties$$
$$\exists P = "Durability" \in PS(A) \Rightarrow \exists O \in FS(A)$$
$$or \exists B \in AS(A) \; such \quad that \; \exists P' = "Durability" \in PS(B)$$

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 applications of a specific domain). 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 structural point of view, whereas 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 [19].

A. 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 [19] specification specifies. Table I shows how the complete specification of a stereotype looks like.

Using the same pattern, we have defined a set of appropriate stereotypes to represent all concepts that are needed for transaction design. Table II shows all the defined stereotypes for the Organization Model:

TABLE I
An example of stereotype definition

Stereotype	«Activity»	
Base Class	Class	
Parent	N/A	
Description	…	
Constraints	None	
Tags	Name	String
	TriggeredBy	String <user, appLogic>
	isSimple	Boolean
	isStrict	Boolean
	isSynchronous	Boolean
	TimeOut	Integer
	mSet	String <e.g. suspend, etc.>
	iSet	String <e.g. begin, etc.>
	tSet	String <e.g. commit, etc.>
	Documentation	Text
Notation	A UML class shape stereotyped as «Activity»	

B. Execution Model stereotypes

In the Execution Model each activity previously defined in the Organization Model, is represented as a state (composite or simple). However, some other pseudostates are used to describe transitive tasks of the application. That is, tasks that do not correspond to user defined activities. Such tasks may be the commit, rollback tasks etc. Table III shows the stereotypes that have been defined.

TABLE II
Stereotypes 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_ctivity»	Class	«ACD_Compensation»	Class
«AI_Activity»	Class	«ACID_Compensation»	Class
«AD_Activity»	Class	«Requires»	Association
«DI_Activity»	Class	«Invisible»	Association
«ADI_Activity»	Class	«Visible»	Association
«ACD_Activity»	Class	«Vital_Visible»	Association
		«Vital_Invisible»	Association

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 must start or terminate the optional activity.

TABLE III
Stereotypes defined for the execution model

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

6. Examples

UTML is capable to describe transactions conforming to the most known transaction models (nested, open nested, sagas, ConTracts, etc.). In parts A and B of this section 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 part C we provide an extended example with custom transactions that are enough flexible and incorporate different behavioral patterns. The example also shows how UTML facilitates a bottom-up design process.

A. 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. In UTML a transaction conforming to this model is described in figure 2.

Table IV shows the main specification properties (as they have been defined in the corresponding stereotypes)

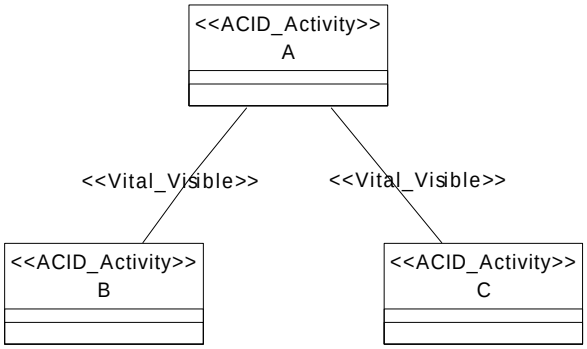

Figure 2: Nested transaction model

of each one of the activities A, B, and C.

Of course the whole specification of the above activities includes many other properties. However, the

TABLE IV
Activity Specification for the nested transaction

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

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 a1. *Activity A cannot complete unless its sub-activities have been previously completed successfully.*

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

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

- **Rule a1**. *Activities B and C can start only after the activity A has started.*

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

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

$\exists O'' = "begin" \in OH(A)$

B. 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 include compensation of other successfully completed sub-transactions. Figure 2 shows the description of a transaction conforming to the Open Nested Transaction model.

Table V shows the main specification properties of

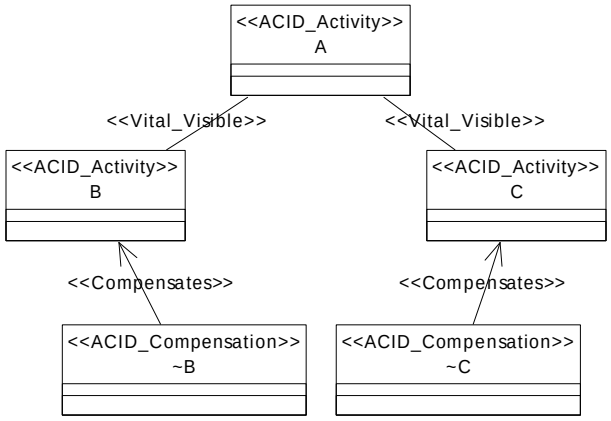

Figure 3: Open nested transaction model

each activity. In Open Nested Transactions sub-transactions can commit prior to the commitment of their parent. In addition, as soon as a sub-transaction commits, it makes its results visible to the outside world. Thus, in case of failure, all committed sub-transactions must be compensated. In UTML each such sub-transaction is associated with a compensation of the appropriate type.

TABLE V
Specification of activities A, B and C

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

Table VI shows specification of the compensating activities ~B and ~C.

TABLE VI
Specification of compensations ~B and ~C

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

The complete specification of Open Nested Transactions includes three well-formedness rules which describe constraints on the behavior of each involved activity, as well as constraints on the whole structure. The following rules have been defined.

Rule b1. *Activity A can commit only after the commitment of the activities B and C.*

$$O, O', O'' : operations; A, B, C : activities$$
$$\exists O \in OH(A) \mid O = "Commit" \Rightarrow \exists O' = "commit" \in OH(B) \wedge \exists O'' = "commit" \in OH(C)$$

Rule b2. *Activities B and C can start only after the initialization of activity A.*

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

Rule b3. *The rolling back of the activity A includes the compensation of any successfully executed sub-activity of A.*

$$O : operation; A, A_i : activities$$
$$\exists O = "abort" \in OH(A) \Rightarrow \forall A_i \in AH(A) \exists \sim A_i \in AH(A)$$

C. Custom transaction – the hotel reservation 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 somewhat involved and describes some aspects (although 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). 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, regardless of 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. In this example, User Authorization is an atomic activity. 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 information system of the bank 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 will show only the UTML diagrams that show the language's ability to describe complex activity organizations facilitating the design process.

C.I. Organization Model diagrams

The following diagrams describe the static structure of the whole activity organization. Each diagram describes the decomposition of an activity:

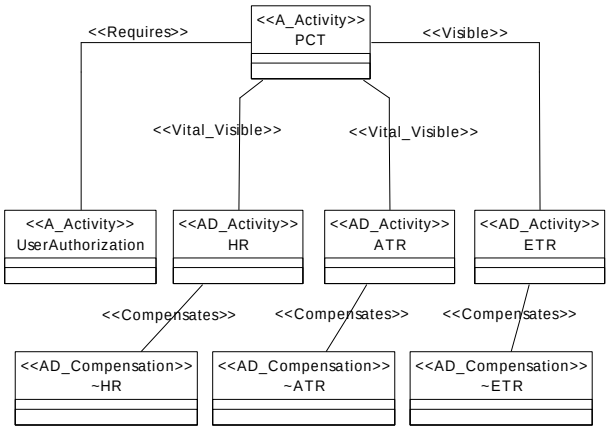

Figure 4: "Plan Conference Trip" activity structure

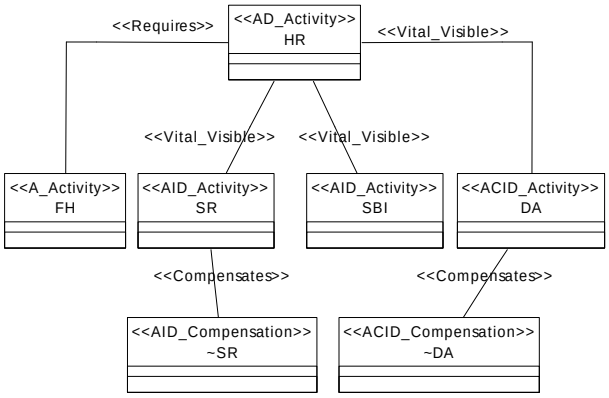

Figure 5: "Hotel Reservation" activity structure

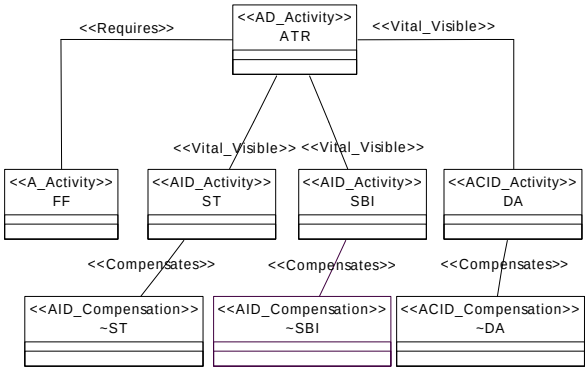

Figure 6: "Airline Ticket Reservation" activity structure

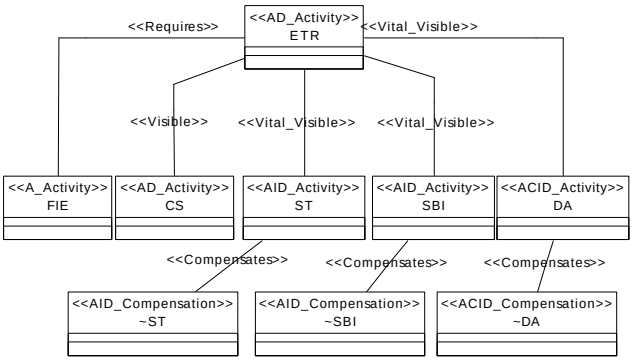

Figure 7: "Event Ticket Reservation" activity structure

C.II. Execution Model diagrams

The following diagrams describe the execution flow for the Plan Conference Trip top-level activity and all its sub-activities:

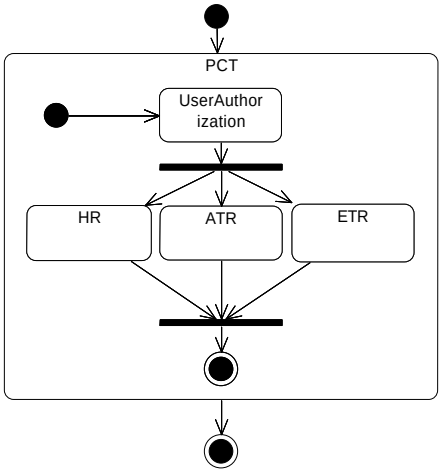

Figure 8: "Event Ticket Reservation" execution flow.

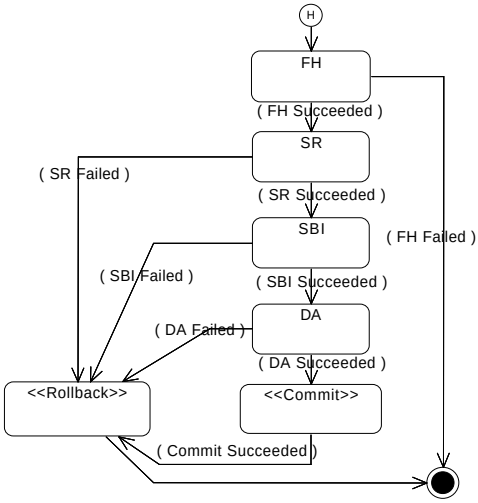

Figure 9: "Hotel Reservation" execution flow.

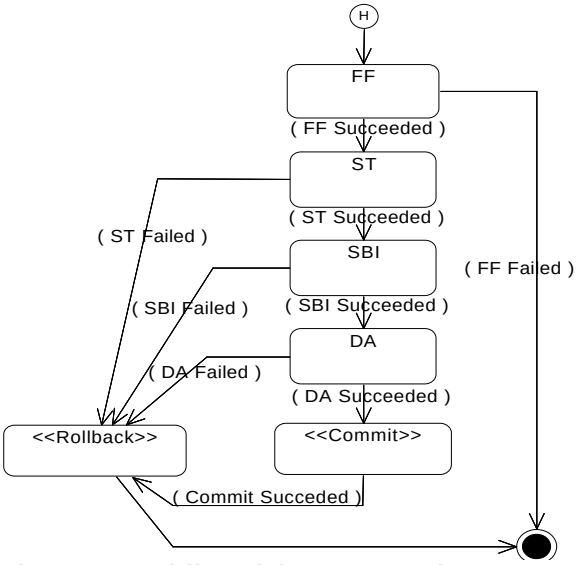

Figure 10: "Airline Ticket Reservation" execution flow

In this example we presented the flexibility that UTML

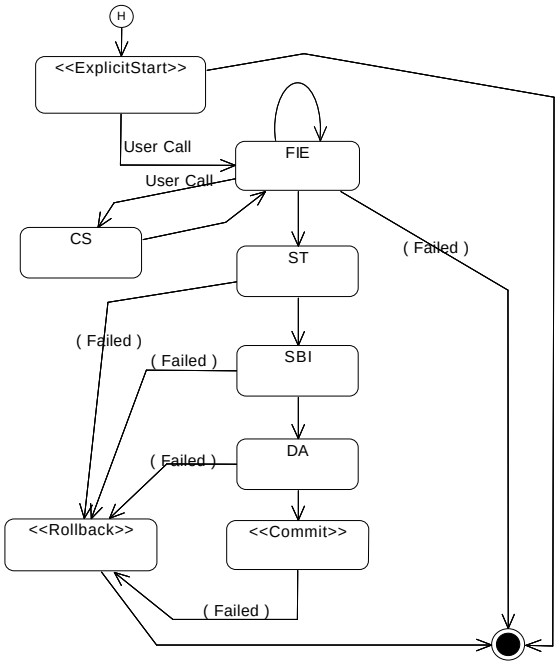

Figure 11: "Event Ticket Reservation" execution flow.

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 very complex task that has to take into account diverse pre-existing resources, 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 impose as few restrictions 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. A tool that allows modeling of transactions based on UTML has been implemented and it is expected to be integrated with a set of other tools (for requirements, customization, and hypermedia design) that the UWA partners will provide shortly. These tools will be

completely evaluated against bank and e-commerce applications.

We are currently working to extend the core UTML functionality in order to include higher level design help for mobile transactions and for common transaction models used in business environments.

8. References

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

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

[3] A. Cockburn: "Writing Effective Use Cases" 2001

[4] C. Pu, G.E. Kaiser and N. Hutchinson, Split-Transactions for Open-Ended Activities, In the Proceedings of the 14th Conf. on VLDB, Morgan Kaufman pubs, 1998

[5] 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

[6] 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

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

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

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

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

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

[12] I. Jacobson, Grady Booch and James Rumbaugh: "The Unified Software Development Process", 1998

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

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

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

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

[17] Object Management Group, Activity Service Specification, www.omg.org, 2001

[18] Object Management Group, Object Transaction Service Specification, www.omg.org, 2001

[19] Object Management Group, Unified Modeling Language Specification 1.4, www.omg.org, 2001

[20] Open Group, Distributed Transaction Processing: XA Specification, X/Open document c193, ISBN 1-85912-057-1

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

[22] 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, 1990

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

[24] R. Barga, D. Lommet, S. Agrawal and T. Baby, Persistent Client-Server Database Sessions, In the Proceedings of EDBT, 2000