UTML: UNIFIED TRANSACTION MODELING LANGUAGE

by

Nektarios Gioldasis

A thesis submitted in fulfillment of the
requirements for the degree of

Master of Computer Engineering

TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING

LABORATORY OF DISTRIBUTED MULTIMEDIA

INFORMATION SYSTEMS AND APPLICATIONS

MUSIC

Chania 2002

DEDICATION

To Christos and Andreas

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis.

I authorize the Technical University of Crete to lend this thesis to others institutions of individuals for the purpose of scholarly research.

I further authorize the Technical University of Crete to reproduce this thesis by photocopying of by other means, in total or part, at the request of other institutions or individuals for the purpose of scholarly research.

ABSTRACT

This thesis proposes UTML (Unified Transaction Modeling Language) as a high level, formal and extensible modeling language for complex transaction models for transactional web applications. Web applications impose several new characteristics that alter the notion of transaction. Such new characteristics are the hierarchical structure of transactions, the dependencies imposed between transactions of the same structure, the distributed nature of resources in the web, the use of other transactional world-wide distributed web services, and the integration (re-use) of diverse resources like legacy systems that already exist in organizations for many years.

This thesis does not propose any new specific transaction model or any new transaction management system. Rather it proposes a design language for transactional web applications that can be used by application designers to analyze, model and document the complex transactions of the web applications and services, as well as to communicate the transactional semantics of an application to any interested party (designers and implementers –current or future ones-, customers or other applications). Many web applications are nowadays delivered to different end users, through different channels and different devices. This "ubiquity" of web applications introduces the need for multiple implementations (or transformations) of essentially the same logic for different devices and channels. We refer to those applications as "families of applications" with several members. Very often, those members are needed and developed at different times during the life time of the family. Thus, proper and formal documentation of the precise semantics that an application family has with a tool like UTML is of high importance.

The language that we developed is based on a rich and extensible transaction meta-model and it proposes a notation system that is used to export the meta-model's functionality in a concrete and formal interface. The meta-model defines the main concepts used in transaction modeling and it regulates their relations and their behavior by introducing a rich set of constraints and rules.

The notation system serves as a graphical tool that makes the use of the meta-model handy and easily understood by any one. The notation system is a compatible extension of the UML (Unified Modeling Language) using its extensibility features, and complements the transaction meta-model by providing modeling of the execution flow of transactions. The use of UML has several advantages such as easy and high level modeling of transactions eliminating the meta-model's complexity, modeling of application's data, logic, and flow of execution with the same language, etc.

The final design of transactions can be exported in XML (eXtensible Markup Language) based on an XML schema that has been properly defined to describe the transactions provided by applications. An XML description of the application's transactions has several uses. One possible is for documentation of the application's functionality and transactional semantics, which can be queried using standard XML query languages. Another is the communication of those semantics between co-operating applications or companies that plan to cooperate for producing an integrated service (for example using ebXML). Finally, by having the application's functionality described in XML format, one can easily transform it to WSDL (Web Services Description Language) documents in case that he wants to make an application (or part of it) available to the outside world as a web service.

The thesis shows that the transaction meta-model is rich enough to describe most of the existing Extended Transaction Models and it presents a complex ubiquitous transactional web application example which can be modeled using UTML. Much more detailed examples of complex web applications and their modeling with UTML can be found in [22].

The implementation of UTML and XML transformation tool have been successfully integrated to a Ubiquitous Web Application design tool, which has been designed and implemented as an extension of UML for facilitating the design of web applications within the European IST project UWA (Ubiquitous Web Applications / IST-2000-25131).

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Prof. Stavros Christodoulakis for his supervision and his continuous guidance throughout my research, as well as for the important experiences he offered me during my stay at the Laboratory of Distributed Multimedia Information Systems and Applications (MUSIC). First of all I would like to thank him for the positive influence he had on me in broadening my horizons.

I would also like to express my gratitude to the readers of this master Mr. Manolis Koubarakis and Mr. Peter Triantafillou for the time they devoted and their critical evaluation and helpful comments on my thesis.

My appreciation goes to Mr. Petros Kapoulas, Nektarios Moumoutzis, Andreas Economides and George Anestis for the endless discussions we had regarding this research.

It is my pleasure to acknowledge Mrs. Maria Koziri, who during her diploma thesis in the lab, she implemented a tool to support the language described in this thesis.

Finally, I would like to thank all my colleagues in Lab so much for our excellent cooperation, as for the pleasant environment in the Laboratory.

PUBLICATIONS

Part of the work that is included in this thesis has been published in the following Conference Proceedings:

- Nektarios Gioldasis, Stavros Christodoulakis and the UWA Consortium, "**The UWA Approach to Modeling Ubiquitous Web Applications**", a paper in the Proceedings of the European Conference on Mobile IST and Wireless Telecommunications, Thessaloniki, June 2002
- Nektarios Gioldasis, Stavros Christodoulakis. "**Transaction Modeling for Web Applications and Services**", in the proceedings of the 1st Hellenic Data Management Symposium (HDMS 02), Athens, July 2002
- Nektarios Gioldasis, "**UTML: Unified Transaction Modeling Language**", a **Doctoral poster** in the Proceedings of 28th International Conference on Very Large DataBases – (VLDB) 2002, Hong Kong - China, August 2002
- Nektarios Gioldasis, Stavros Christodoulakis and the UWA Consortium, "**Ubiquitous Web Applications**", In the proceedings of the 12th European Conference on e-Business and e-Work (EBEW), Prague, October 2002
- Nektarios Gioldasis, Stavros Christodoulakis, "**UTML: Unified Transaction Modeling Languag*e***", In the proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE), Singapore, December 2002

TABLE OF CONTENTS

LIST OF FIGURES

Figure *Page Number*

LIST OF TABLES

1. Introduction

The concept of transactional computation is not new. Decades ago people begun thinking of requirements and constraints that should be enforced by applications' logic and computers' infrastructure, in order to prevent wrong processing or handling of critical data. The field of applicability was initially identified to be in banking applications. The requirements identified for those applications led to the characterization of some programs as ***transactions.*** That is, a transaction is a program. Fortunately, the transaction concept has been defined to be independent to the banking applications and thus, its use proved valuable in many application domains beyond this area. So, in this thesis transactions are considered and presented as general programs that are enhanced with additional semantics.

1.1. What makes a program a transaction?

In many applications databases are used to model the state of some real-world enterprise. In such applications, transaction is a program that interacts with the database so as to maintain the correspondence between the state of the enterprise and the state of the database. In particular, a transaction may update the database such as to reflect the occurrence of some real-world event that affects the state of the enterprise. An example is a reservation of a seat at particular flight of an airline. The event is that a customer reserves a ticket for a particular flight with this airline. The transaction updates the airline's database to reflect the reservation of this particular seat.

Transactions however, are not ordinary programs. Requirements analysis for transactions has shown that they should obey specific constraints which distinguish them from other non-transactional programs. The general idea behind the transaction concept is that a transaction is a contract. In making a contract, two or more parties negotiate for a while and then make a deal. This deal imposes some constraints that should be obeyed by all involved parties. These constraints often are:

Atomicity: *A transaction either happens or it does not; either all defined work is bound by the contract or nothing (as it had never been started).* In other words, transactions either complete successfully or any partial result of them is undone and the database is not affected at al. In general, the status of a transaction after its termination can be either ***committed*** (all work completed successfully) or ***aborted*** (no actions were executed and no partial effects have survived in the system). This constraint (or property that must be supported by transactions) is also known as failure atomicity. That is, in case of a failure, all active transactions should abort and all so far updates that they have performed to the database should be undone (rolled back) so that the database obtains the state that it had before the aborted transactions had been started. The process of restoring the database in the state it had before an aborted transaction started is part of the ***database recovery*** and it is executed by the database recovery manager.

Consistency: *Transactions should transfer the database from one consistent state to another consistent state.* A transaction must access and update the database in such a way that it preserves all database integrity constraints. Every real-world enterprise is organized in accordance with certain business rules that restrict the possible states of the enterprise. For example the airline business rules define that the number of reserved tickets for a flight must be less than the number of the total seats for this flight. As it has been mentioned, databases are used to model real-world enterprises and thus, when such business rules exist, they have a finite number of states.

In database terms, these rules are stated as integrity constraints. The integrity constraint corresponding to the above business rule asserts that the value of the database item that stores the reserved seats for a flight cannot exceed the value of the database item which stores the number of the total seats for this flight. Thus, when a transaction on this database terminates, the system must ensure that all database integrity constraints are satisfied. Otherwise, the transaction is considered to break the contract and is not allowed to commit (to terminate successfully); it should abort by rolling back (undoing) all its partial results.

Transaction consistency is checked at the termination of a transaction. This holds due to the fact that a transaction performs database operations (reads and writes) sequentially as a normal program. Thus, during the transaction processing, inconsistencies in the database are possible. However, the definition of consistency imposed that transactions should see a consistent state of the database. Thus, these inconsistencies should not be visible by other transactions on the same data. This means that transactions should not interfere.

Isolation: *Even though transactions are executed concurrently, the overall effect of the schedule must be the same as if the transactions had executed serially in some order.* We say that a set of transactions are executed sequentially, or serially, if one transaction of this set is completed before another transactions is started. In this case, if all transactions are consistent and the database was initially in a consistent state, then after the execution of all transactions the database is still consistent. However, serial execution of transactions is impossible for applications that have strict performance requirements. For those applications, concurrent execution of transactions is the only way to meet the performance requirements. Concurrent execution is appropriate for systems that serve many users and at any given time, and in those systems it is possible that many partially completed transactions will be active in the same time.

In concurrent execution, the database operations of different transactions are effectively interleaved. Operations are sent by a transaction to the database management system (DBMS) forming a sequence of requests. Such a sequence is known as a ***transaction schedule***. When transactions are executed concurrently, the overall database schedule is simply a merge of the schedules of all active transactions. If the DBMS serves these requests with the sequence that they arrive, then it is possible that transactions will see partial results of other transactions (that may or may not abort in the future) and based on these results may perform database operations that violate the database consistency. An unacceptable situation! Thus, appropriate constraints must regulate the concurrent execution of transactions. In particular, constraints that prohibit transactions interference must be provided and the transaction processing system must enforce those constraints.

One oversimplified approach is to have transactions to be executed serially. However, this approach is inapplicable in multi-user systems where performance requirements are strict. On the other hand, it would be a waste of resources (and time) if we had an active transaction blocking another transaction that would like to perform operations on different database objects. Thus, transactions should be able to execute concurrently with other transactions if they do not interfere. Transaction schedules that satisfy this constraint are called **serializable**.

Isolation is usually achieved by requiring transactions to obtain locks on the database items on which they want to perform operations. When a transaction T_1 asks for a lock on a database item that is held by another transaction T_2, then T_1 has to wait until T_2 releases the lock on that item. The strategy that will be used to enforce isolation is known as the ***concurrency control*** and the DBMS module that is responsible for that, is called ***concurrency manager***. There have been developed several concurrency controls that are implemented by DBMSs.

As with Atomicity, and Consistency, ordinary programs do not necessarily have to obey the constraint of isolation. For example, if common programs that update a particular file are executed concurrently, updates may be interleaved and the overall output (the file content) may be quite different from that obtained if they had been executed sequentially.

Durability: *Once the results of a transaction have been committed, they cannot be aborted by the same transaction.* This requirement imposes that once the transaction commits, the system must ensure that its effects remain in the database even if the computer, or the medium, on which the database is stored, crashes. Consider for example that you are a client that you want to reserve a seat in a flight. Once you have reserved your seat, you require that at the boarding time you will still have the seat reserved for you, independently of any failure in the airline's information system.

On the other hand, real world deals can be canceled. Thus, committed transactions should be still able to be canceled. This is of course possible, but a committed transaction can be only canceled by another transaction; not the same.

Durability against physical failures can be achieved by several ways. Backups, replication at mirror sites, etc. This has to do with the degree of data availability that we desire, and the cost of ensuring durability varies with the desired degree of availability.

1.2. Advanced Transaction Models

In section 1.1 we described transactions to be executed on a single computer and access a single database. Transactions of this kind are known as ***database transactions*** and are defined to be short in duration, flat –without internal structure- and access a single database. Their analysis and modeling has been done in this context and most DBMSs support effective processing of them.

Although powerful, the transaction model adopted in traditional database systems is found lacking in functionality and performance when used for applications that involve re-active (endless), open-ended (long-lived) and collaborative (interactive) activities. Hence, various extensions to the traditional transaction model have been proposed and are known as *Extended Transaction Models (ETM)*. ETMs try to relax some of the ACID properties that the traditional transaction model enforces by successively decomposing a complex transaction into sub-transactions in a top-down fashion. Each ETM defines a specific transactional behavior, and all nodes (transactions) of a complex transactional graph have to follow this behavior.

1.3. The Motivation for UTML

In the fist years of the internet and the world-wide-web (www), enterprises were provided with the ability to promote themselves by presenting information to their potential customers. Nowadays, the internet's evolution is so high, that applications have the ability to deliver their functionality to a wide range of users and even other applications through the web. The term ***e-business*** has become a buzzword and refers to the new way with which enterprises make business. The internet-provided interconnectivity between applications led to the reconsideration of the traditional notion of enterprises and consequently the notion of enterprise information systems. From a transactional point of view, ***web-based information systems or web applications*** are very complex and impose several issues coming from different sources.

1.3.1. Complex User Interface Interaction

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 (probably trained) 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.

While it is possible to place several restrictions in the user navigational patterns (including restrictions on the use of the web browser capabilities), such restrictions should be as limited as possible in order 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 consequently **web 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***.

1.3.2. Distributed and Diverse Resources

Transactional web applications may be composed of several hierarchically structured activities that may access distributed resources. If such activities are defined to have transactional semantics, then new requirements appear that have to be taken into account by both the application designer and developer.

Think of a web application that, among others, offers the functionality of performing cross-bank money transfers. Suppose that this process is implemented with a distributed transaction *T*. This transaction initiates two other sub-transactions T_1 and T_2, one at each bank site. The global transaction must ensure that either both T_1 and T_2 commit, or none commits. An illustration of this example is depicted on figure 1.

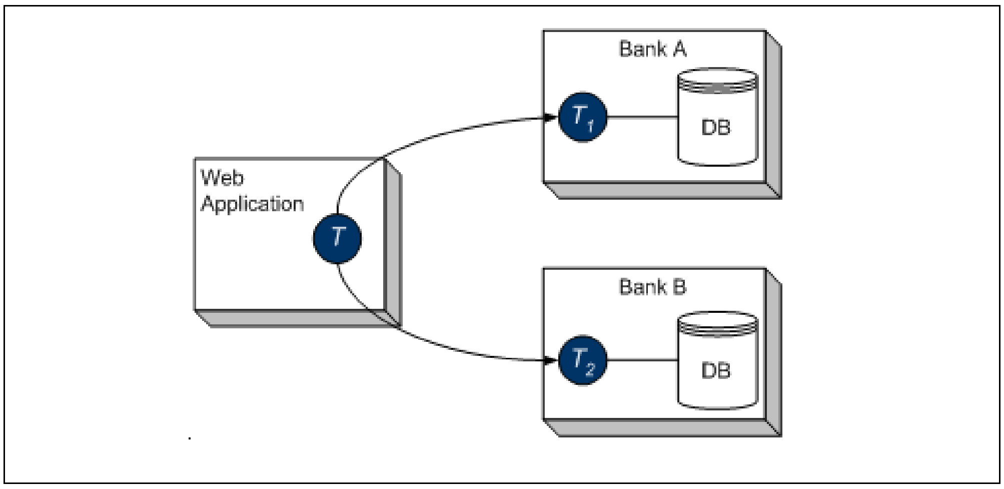


Figure 1: A distributed Transaction

In this example, except that each (local) sub-transaction must be atomic, consistent, isolated and durable, the whole distributed transaction T should be globally atomic and consistent. That is, either both T_1 and T_2 commit, or none of them commits. The problem derives from the distributed nature that the application has. If both accounts were held in the same database, then the business process would be modeled by one traditional transaction and the problem wouldn't exist.

To overcome this problem, proper commit protocols had to be defined. The Two Phase Commit (2PC) does it by defining a standard interface which has to be supported by both DBMSs in order to communicate and co-ordinate the commitment of distributed transaction.

To make things worse, suppose that one distributed DBMSs does not support the 2PC protocol. How is it now ensured that either all distributed transactions commit, or none commits? As another example of transactions that accesses

diverse resources consider one that updates both a database (enforced integrity constraints, isolation, etc.) and a file handled by an ordinary file system.

Thus a web transaction model should be flexible enough to accommodate diverse resources into the scope of the same structured transaction, taking into account different resource interfaces and semantics.

1.3.3. Distributed services and legacy systems

The internet enabled traditional information systems to deliver their functionality through the www to end users and other applications. Web based information systems may be built from scratch or may exploit functionality that already existed, and the enterprises have tested for years and they trust them. On the other hand, a web based information system may be a combination of both existing functionality and new that complements the existed one.

Much of the pre-existing functionality has been built many years ego, when technology was no so advanced, and its poor documentation and high complexity makes its modification extremely difficult and dangerous. The term ***legacy system*** refers to those systems that have the aforementioned characteristics.

From a transactional point of view, when trying to integrate legacy systems into web applications there are two main problems:

- **Simplified data manipulation:** Many legacy systems process their data without the ACID transaction model in mind. Their correct use may require a good training on their functionality and a deep knowledge of their limitations. In developing new web-based applications, this is an important factor. Recall that users of web applications may be completely inexperienced.

- **Poor documentation:** Legacy systems have been developed without integration in mind and the help of their implementers may not be available at the time of integration to web applications. Moreover, most of times there is no documentation available to the designers and implementers of the new integrating web application.

Web transactions that are defined to utilize functionality of legacy systems should be designed in a bottom-up fashion, taking into account all limitations that come from this integration. Existing transaction models do not properly support bottom-up design and thus, their use in modeling web transactions proves to be difficult or even inapplicable. ***A new –web oriented- transaction model is needed that will be able to support transaction design in both a top-down and bottom-up fashion.***

Another example of pre-existing functionality exploitation by web applications concerns the integration of ***web services***. The emergence of XML (eXtensible Markup Language) has fired an entirely new area of e-business. Interoperability between applications became possible through XML messages. This ability enabled the technology of web services, through which the functionality of a web application or an information system (possible a legacy one) can be available for other applications as a (web based) service.

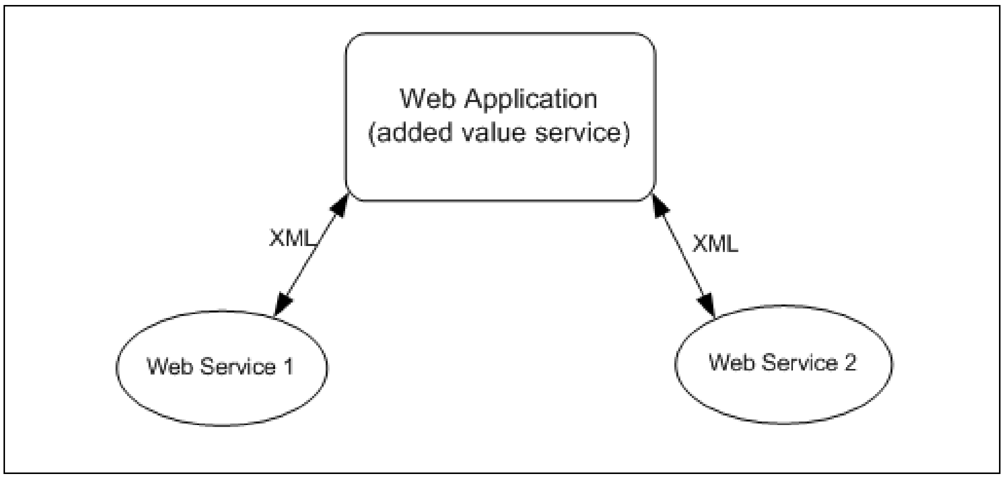


Figure 2: A Web Application Utilizing Web Services

In a web application, remote web services may be used as activities or sub-activities that specify a specific user goal or sub-goal. Figure 1 depicts such a web application. In case that the application logic implies dependencies between these services and other newly developed transactional activities of the application, no one of the existing transaction models can be used to exactly describe these dependencies. Thus, ***a web transaction model should provide proper integration of web services into a web application, taking into account the transactional logic of these services and offering added value services to the end user.***

1.3.4. Ubiquity

Web applications may be presented to the users in diverse devices (mobiles, palmtops, etc.) having deviations in the interface and the flow of logic, thus forming families of applications that utilize the same application logic with rather small deviations.

Consider for example a web application that offers the functionality of reserving flight tickets. If this application is to be delivered through a mobile phone, then the implementation of the new «application view» will use the already existing transactional logic of the application. In such a case, the transactional logic and the precise semantics of such applications should be ***well modeled*** in order to be reusable. Moreover, not all «application views» are known and taken into account during the application design and implementation. So, ***there is a need for proper documentation of the application's logic and the precise transactional semantics,*** in order for the new application views to be easily derived.

The above requirements for web applications show that web transactions can be very complex for well designed applications, and that flexible transaction models and 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[26], ***there are no such mechanisms*** to facilitate the modeling and design of application logic that exhibits complex transactional behavior.

1.4. The Scope of the Thesis

This thesis brings together high level modeling mechanisms and transactional aspects of application logic. It proposes UTML (Unified Transaction Modeling Language) as a high level transaction modeling language to facilitate the complex transaction design process of web applications. 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. It allows great flexibility for the web user navigational patterns, and also accommodates long-lived transactions. UTML is an extension of the Unified Modeling Language [26] (the most widely accepted and used industrial modeling standard) and supports the ***modeling, documenting and maintaining*** of large scale transactional web information systems.

1.5. Aim, Objectives, and Contribution of the thesis

The aim of this thesis is to provide a ***high level modeling language*** for web applications that exhibit complex 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 application views of the same application family (like an application re-implemented or transformed several times for different terminal devices).

The Unified Modeling Language (UML) has been chosen as the modeling platform on which the proposed language (UTML) has been built. UML is a world-wide industry standard for modeling, and UTML has been built on top of it using its extensibility mechanisms and consequently is completely compatible with it.

UTML should not simply extend UML by defining some new stereotyped model elements. Rather, it should bring together a flexible, rich, and extensible transaction meta-model with the high level modeling mechanisms provided by UML, forming a powerful high level transaction modeling language capable to support the design of complex web transactions.

The detailed objectives that have been set for UTML derive from the limitation that existing transaction models have and can be summarized into the following:

- Give to the designer the ability to ***analyze, design, and describe both the static structure of transactions and their dynamic behavior***. In web applications, it is very 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). This way, transaction execution is smoothly integrated to the entire application and a primitive user navigation model is defined.

- Provide appropriate ***tools for designing transactions compatible to most of the known transaction models***. The existing transaction models should be always a choice of design. Thus UTML must provide mechanisms to design transactions that conform to these models.

- Provide extensibility for ***describing new transaction models*** that may be required according to the application's requirements. No specific model can capture the requirements of any web applications. Thus, it is important for UTML to be extensible, giving the ability to dynamically synthesize new transaction models as web applications require.

- Provide the ability for ***describing different decomposition semantics and behavior into the same structured transaction***. This flexibility is very important for applications that access resources (databases, legacy systems, file systems, etc.) with different interfaces, behavior and semantics. 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.

- 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).

UTML has been developed to meet the aforementioned objectives and provides a formal mechanism for designing complex web transactions. To my knowledge, UTML is the first transaction modeling language in the literature and is compatible to the Unified Modeling Language. Beyond the achievement of the objectives that were previously described, it provides a complete set of rules that are used to formalize the arbitrary structuring of transactions in complex transactional graphs. Alike other models, it can be used to describe "weak transactions", i.e. activities that do not have to respect the entire set of the ACID properties, providing this way a mechanism for modeling typical web processes (that are not strict transactions) and integrating pre-existing functionality of legacy systems that do not ensure all the ACID properties in transaction processing.

In [22] a design tool has been developed in order to support transaction design with UTML. This tool enforces the application of all well-formedness rules that the language proposes and provides the ability of describing the transaction design in XML format. This capability enables the communication of transactional semantics between interacting applications and provides easy description of well-designed applications as transactional web services.

1.6. The Tourist Support System

To enhance reader's understanding of the technical material presented here, a case study of a transactional web application is being used through the chapters of this thesis. This case study is a Tourist Support System (TSS). This application provides a complete functionality to tourists (end users) in organizing their vacations. Users can register to the system and use its full functionality by making flight reservations, hotel reservations and reservations of tickets for some social events. In this system there are several tasks that can be executed by the users. However, three of them will be used, when needed, to enhance the reader's understanding. A brief description of these tasks is needed to familiarize the reader with the TSS, which will be mentioned many times in this thesis.

- **User Authorization.** In order for the users to access the functionality of TSS they have to register to the system. Once registered, each time that they want to use the system's functionality they have to authorize themselves. User authorization is a way for the system to identify the user. Even in cases that the use of an application's functionality is free, web applications do still require users to authorize themselves (in order to provide personalized functionality, content, etc.). In such a task usually some private user data are required, and a validation of those data against all stored user profiles is done.

- **Flight Reservation.** With this task users can reserve tickets in any flight and pay the cost of the ticket. A particular flight reservation requires several actions as to find a desired flight, to check availability, to reserve ticket and to pay the ticket's cost.

- **Hotel Reservation.** Users execute this task to make hotel reservations according to their requirements. Hotel reservation may depend on the output of the flight reservation task or vice versa. That is, a user may want to make a hotel reservation in Rome if he manages to make a flight reservation for going to Rome. In case that he can find a flight he may wish to cancel the hotel reservation since there is no way to go there. In order for a user to make a hotel reservation he should find a hotel, find a room, reserve the room and pay the reservation cost.

- **Event ticket reservation.** Users may or may not want to reserve tickets for some social events that may take place during his residence in Rome. If yes, he has to find interesting events, to check availability of tickets and reserve the tickets he desires. Finally he must pay the cost of the tickets that he reserved.

The tasks that were described above can be executed independently of each other, or under the scope of a large, long-lived, complex task of planning a trip. More details about those tasks will be presented through the chapters of this thesis.

1.7. Thesis' structure

Into the next chapter we briefly present the work which is related to this research. In particular we present the most known Extended Transaction Models in the literature and we identify their limitations in relation to complex transactional web applications. Also, we give a brief presentation on the Unified Modeling Language.

In chapter 3 we present the transaction meta-model on which UTML has built on. The meta-model sets up the formal concepts used in designing complex transactions and defines the modeling elements used in UTML. The concepts of operations, activities, activity execution contracts, compensations, well-formedness rules, and well-behaving rules are presented in detail, as well as, the extensibility mechanism of the transaction meta-model.

Chapter 4 describes the notation system of UTML. The notation system is an extension of the UML which has been done by using its extensibility mechanism.

It consists of two parts: the Organization Model and the Execution Model. The former is used to specify the structure of complex transactions and their decomposition semantics, while the letter one is used to define the flow of execution between them.

In chapter 5 we present the transformation of transaction design from UTML to XML. This transformation is part of a design tool, implemented outside the scope of this thesis, which supports transaction design with UTML. In particular, we present the XML schema, which has been developed in the scope of the thesis and is used to produce XML documentation of the transaction design.

In chapter 6 we demonstrate the use of UTML in designing complex transactions. We firstly use UTML to describe known transaction models (Nested and Sagas), and then we provide an extended example on designing custom transaction models with this language. The example used in that chapter is the Tourist Support System, and it demonstrates the flexibility of the language in describing complex transactions for web applications.

Finally, in chapter 7 we conclude the work presented in this thesis, and we summarize its main contributions. We also present some ideas for future extensions of UTML to directions of dataflow modeling between transactions, modeling of persistent transactions, and description of asynchronous transaction execution.

2. RELATED WORK

In this chapter, we present the work that is directly or indirectly related to UTML. One research area that UTML is directly related to, concerns the Extended Transaction Models and distributed transaction management standards and protocols. The other area concerns the Unified Modeling Language research efforts.

In section 2.1 we present the most known Extended Transaction Models and we identify their limitations in relation to complex web transactions. The Extended Transaction Models will be presented are:

- **The Nested Transactions Model**
- **The Open Nested Transactions Model**
- **The Sagas Transaction Model**
- **The ConTract Transaction Model**
- **The Split Transaction Model**
- **The ACTA Transaction Framework**

Then, in section 2.2 we give a brief presentation on UML, in order to familiarize the user with the concepts that will be used in the following chapters of this thesis. UML is a world-wide accepted industry standard, which is used for analyzing, modeling, and documenting software systems.

CHAPTER 2. RELATED WORK

2.1. Extended Transaction Models

The insufficient expressiveness of the traditional transaction model led to the development of many new, so called *extended transaction models*. These models try to overcome the limitations of the traditional model by, for example, introducing the internal structure of a transaction, or by relaxing some of the ACID properties. Relaxed atomicity for example, allows a transaction to succeed (commit) even if some of its operations or sub-transactions fail, and thus provides finer granularity of recovery control. Relaxed isolation allows a transaction to reveal its partial results to a concurrent transaction, and thus increases possible level of concurrency and cooperation among the transactions.

In the next sections, the most influential extended transaction models are described. After that, we discuss their limitations when used to describe transactional behavior of modern web applications.

2.1.1. Sagas

Sagas have been proposed by Garcia-Molina and Salem [11], as a model for long-lived transactions. A saga is a set of relatively independent sub-transactions denoted $T_1 \ldots T_n$. The component sub-transactions have ***all the ACID properties*** of traditional transactions, and can interleave in any way with component sub-transactions of other sagas. When a component transaction terminates, it commits and makes its results visible to other sagas. However, sub-transactions of a saga have to be executed in a predefined order.

For each sub-transaction T_k ($1 \leq k \geq n$), a *compensating sub-transaction* CT_k is defined. A compensating transaction CT_k semantically "undoes" the effects of transaction T_k. The state of the database after executing the sequence T_k, CT_k should be the same as if neither T_k nor CT_k were executed.

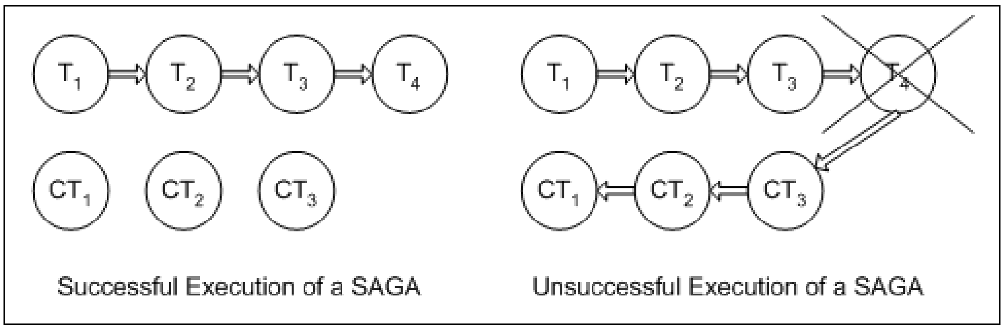

T1
T2
T3
T4
CT1
CT2
CT3
Successful Execution of a SAGA
T1
T2
T3
T4
CT1
CT2
CT3
Unsuccessful Execution of a SAGA

Figure 3: An Example of a SAGA

To complete a saga, either the whole sequence is successfully executed (figure 3 - successful execution of a saga) or the effects of already committed sub-transactions are undone by a sequence of compensating sub-transactions (figure 3 - unsuccessful execution of a saga). As illustrated in figure 3, compensating sub-transactions are executed in reverse order of the component sub-transactions. Note that there is no compensation sub-transaction associated with the last sub-transaction T_n. When T_n commits, no other sub-transaction may be executed (saga commits). Therefore compensating actions for T_n are not required.

The main property of sagas is that their isolation is limited to the level of sub-transactions. Each sub-transaction commits and releases resources. Therefore, sagas can use partial results of other sagas. Clearly, the execution of sagas does not use serializability as a correctness criterion. This happens, due to the fact that sagas may read data that have been updated by other sagas and which may be compensated later on.

Execution of sagas is characterized by an increased degree of parallelism. The resources held by a sub-transaction, and its results are released immediately after its commitment, without waiting for the completion of other components of the saga.

2.1.2. Nested Transactions

To overcome the limitations of the traditional, *flat* transaction model, where a transaction is an atomic unit without any interval structure, *nested transactions* were proposed by Moss in [21]. A transaction in this model consists of several sub-transactions, which in turn may contain any number of sub-transactions, forming a hierarchy called a *transaction tree*. A sub-transaction which has its own sub-transactions is called a *parent*, and its sub-transactions are called *children*.

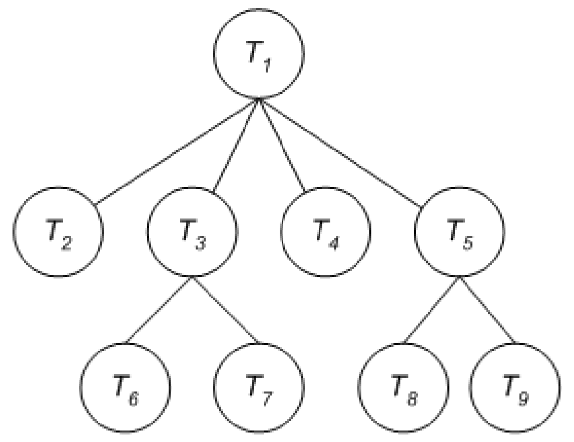


Figure 4: Structure of Nested Transaction

The sub-transactions of a nested transaction may commit or abort independently, subject to the following constraints. A child sub-transaction must start after its parent starts. A parent must terminate only after all its children terminate. If a parent is aborted, all its children must be aborted. However, when a child transaction fails, the parent may choose its own way of recovery. For example, the parent may:

- Ignore the failure and proceed with other tasks. In this case the failed child is considered to be *non-vital*
- Retry the failed sub-transaction,
- Execute another sub-transaction that performs an alternative action (a *contingency sub-transaction*)
- Abort.

All the traditional ACID properties are preserved in this model. Nested transactions are isolated from each other and in case of failure they are rolled back without effects upon other transactions or the database system. However, the sub-transactions of a nested transaction, though atomic and isolated from each other, are not durable. Even if a sub-transaction commits, its effects will be undone when its parent aborts. The updates made by a sub-transaction become permanent only after the root of the transaction tree commits. Similarly, the results of a committed sub-transaction may be used by other sub-transactions before their parent commits, but they are externalized only after commitment of the whole transaction.

The main advantages of this model are:

- **Increased modularity.** The transaction tree provides a convenient framework for hierarchical decomposition of a transaction.
- **Better failure handling.** Sub-transactions allow the users to define recovery units much smaller than the whole transaction. In case of a failure, only a small portion of the performed activity (a sub-transaction) has to be rolled back. In contrast, in the traditional transaction model the

whole transaction must be undone. Such flexibility may be used in developing more efficient recovery mechanisms.

- **Higher degree of parallelism.** Since sub-transactions reveal their results to each other, they may be executed concurrently. Therefore, nested transactions allow a higher degree of intra-transaction parallelism.

The hierarchical approach of nested transactions, as well as the notions of contingency and non-vital components was incorporated into most of the subsequent transaction models.

2.1.3. Open Nested Transactions

Open Nested Transactions [10] relax the isolation requirement of the regular nested transaction model by making the results of committed sub-transactions visible to other concurrently executing nested transactions. This way, a higher degree of concurrency is achieved. To avoid inconsistency use of the results of committed sub-transactions, only those sub-transactions that *commute* with the committed ones are allowed to use their results. We say that two transactions (or, in general, two operations) commute if their effects, i.e., their output and the final state of the database, are the same regardless of the order in which they were executed. In conventional systems, only *read* operations commute. Based on their semantics, however, one can define also update operations as commutative (for example increment operations of a counter).

This transaction model uses *compensation* to provide correctness of transactions. A sub-transaction can commit and release the resources before the parent transaction successfully completes and commits. If the parent transaction later aborts, its failure atomicity may require that the effects of already committed sub-transactions be undone by executing *compensating sub-transactions*. A compensating sub-transaction $\overline{T}$ semantically *undoes* effects of a committed sub-transaction T, so that the state of the database before and after executing the sequence $T\,\overline{T}$ is the same. However, an inconsistency may occur if other transaction S observes the effects of sub-transactions that will be compensated later [11] [14]. The open nested transaction model uses the commutativity to solve the problem. Since only sub-transactions that commute with committed ones are allowed to access the results, the execution sequence $T\,S\,\overline{T}$ is equivalent to $S\,T\,\overline{T}$ and, according to the definition of compensation, to S, and therefore is consistent.

In addition to the modularity, fine granularity of failure handling, and increased level of intra-transaction parallelism, the open nested transaction model provide the user with relaxed isolation and possibly a higher level of cooperation for his applications.

2.1.4. The ConTract Transaction Model

The basic idea of the ConTract transaction model [15] is to build large applications form short ACID transactions. Its exact definition is:

A ConTract is a consistent and fault tolerant execution of an arbitrary sequence of predefined actions (called steps) according to an explicitly specified flow of control (called script).

Each step of a ConTract is implemented by embedding it into a traditional ACID transaction. Thus, steps have all ACID properties, but the ConTract as a whole does not. The relation of a ConTract, as a unit of work, with the ACID properties has the following deviations:

- **Atomicity.** The fundamental deviation from classical transactions is that ConTracts give up atomicity at the script level since they are used to model long duration units of work. In case of failure they roll forward, maybe along a different path than the one taken before.

- **Consistency.** ConTracts maintain system integrity by providing appropriate semantic dependencies between steps.

- **Isolation.** A ConTract is not isolated since it is used to describe a long-lived process.

- **Durability.** Each step of a ConTract is durable when terminates and in order to by undone, a new step must run.

Figure 5 shows a typical ConTract example. In this example the script of a business trip planning activity is described using short ACID transactions. Note that dependencies between steps can be defined by requiring that either they both commit or none commits. However, it is not described how this atomic commitment of both transactions is managed.

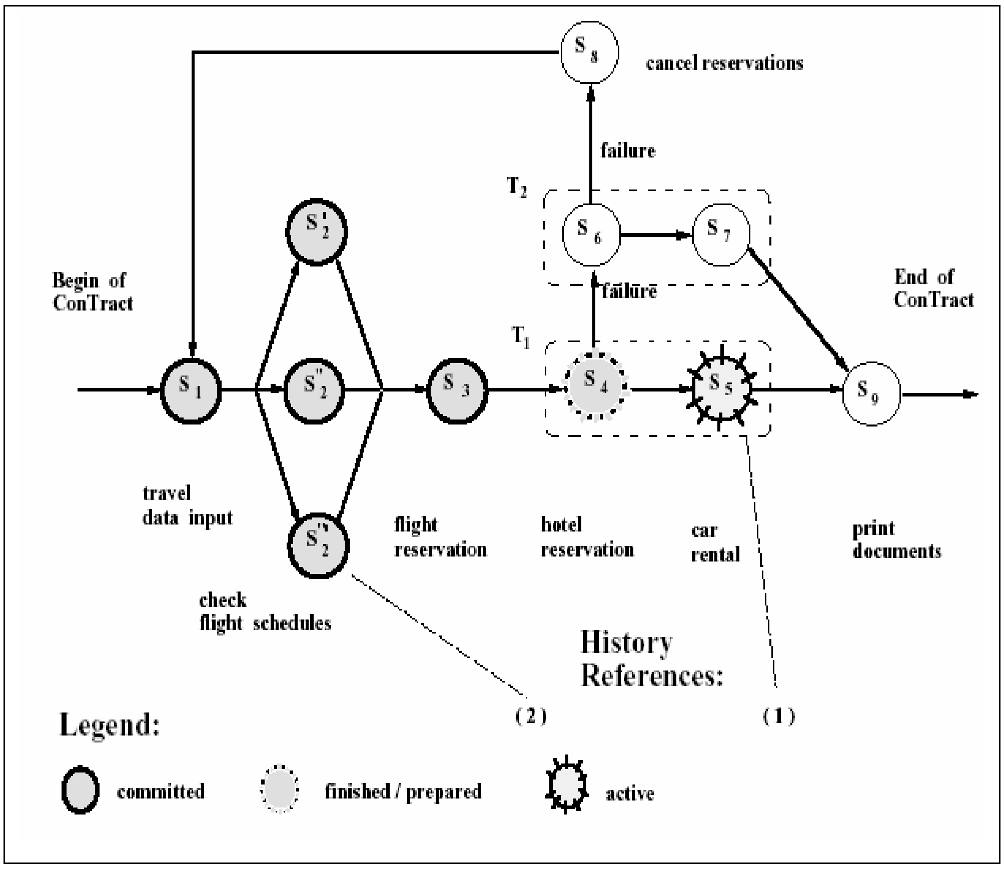


Figure 5: A ConTract Example

2.1.5. *The Split Transaction Model*

In the split transaction model [5], a transaction T_a can split into transactions T_a and T_b. At the time of split, operations invoked by T_a up to the split can be divided between T_a and T_b making each responsible for committing and aborting those operations assigned to them. In order to facilitate further data sharing between T_a and T_b, operations which remain into the responsibility of T_a may be designated as not conflicting with operations invoked by T_b after the split, and hence, T_b can see the effects of these operations.

Depending on whether or not such operations have been designated, a split may be serial, or may be independent. In the former case, T_a must commit in order fro T_b to commit, whereas in the latter case, T_a and T_b can commit independently.

After the split, T_a can split again creating another split transaction T_c. Split transactions can further split creating new split transactions. A sequence of serial splits leads to a different type of hierarchically structured transactions from those of nested transactions.

2.1.6. ACTA Transaction Framework

ACTA was proposed by Chrysanthis and Ramamritham [29][30] as a framework for specifying the structure and behavior of complex applications and for reasoning about their transactional properties. ACTA is not a transaction model itself, rather it is a *framework*, intended to unify existing models and facilitate their analysis. In the ACTA framework, it is possible to characterize the whole spectrum of interactions between transactions, as well as effects of transactions on accessed objects. A taxonomy of the interactions that can be expressed in ACTA is presented in Figure 6.

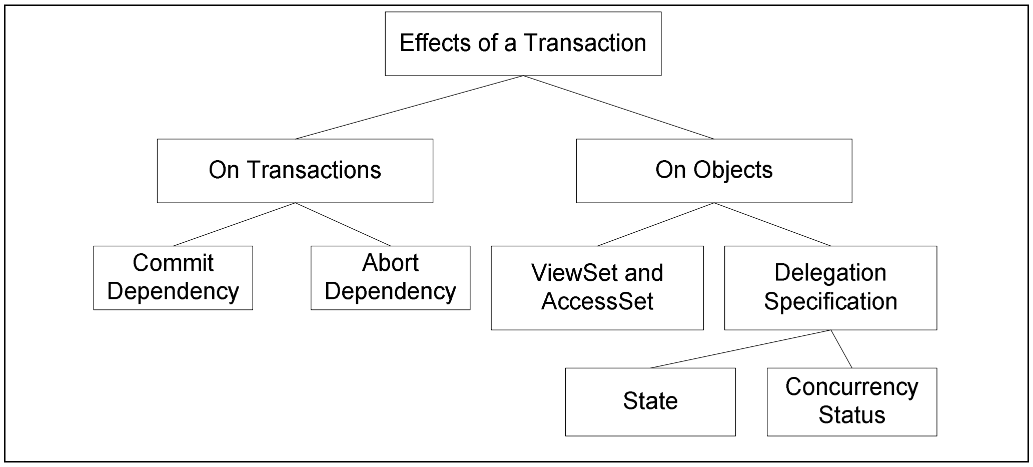


Figure 6: Dependencies Captured by ACTA framework

In ACTA a transaction has two possible outcomes, namely *commitment* and *abortion*. A transaction may develop two dependencies on any other transaction:

- **Commit-dependency:** if a transaction *A* has a commit-dependency on transaction *B*, then transaction *A* cannot commit until transaction *B* either commits or aborts. It does not imply that the two transactions should commit or abort together.

- **Abort-dependency:** if a transaction *A* has an abort-dependency on transaction *B*, and if transaction *B* aborts, than transaction *A* should also abort. It neither implies that if transaction *A* aborts, *B* should abort, nor that if *B* commits, *A* should also commit.

An object accessed by a transaction can be characterized by its state and its status. The state of an object is simply its contents. The state is changed when an operation invoked by a transaction modifies the contents of the object. The status of an object is represented by a synchronization information (e.g., concurrency control information) associated with the object. A timestamp of the last write operation may be an example of the status information. The status of an object changes when a transaction performs an operation on that object.

The effects of transactions on objects are captured in the ACTA model by the concept of *delegation* and by introduction of two sets, the *ViewSet* and the *AccessSet*. The *ViewSet* of a transaction is a set of objects potentially accessible to the transaction. An object from the *ViewSet* of the transaction can be accessed by this transaction only if the concurrency control status permits it. Objects already accessed by the transaction are contained in its *AccessSet*.

A transaction can *delegate* the responsibility of finalizing its effects on some of the objects in its *AccessSet* to another transaction. That is, the delegation represents the ability of a transaction to resign from some of its objects which are taken over by another transaction. Delegation is useful in revealing partial results (delegation of state) and coordination information (delegation of concurrency status) to other transactions. The notion of delegation allows for modeling and reasoning about dynamic transaction models such as, for example, split and join transactions.

The ACTA framework may be useful in better understanding the nature of interactions between transactions and the effects of transactions and improve their concurrency and recovery properties. It makes easier the development and analysis of new extended transaction models suited for a particular environment. However, not all properties of transaction models can be captured and expressed in ACTA, and when an attempt is made to define a transaction with a particular set of properties, the ACTA framework proves very difficult to use.

2.1.7. Limitations of ETMs

The Extended Transaction Models that were previously described are the most known ones in the literature. In this section we discuss their limitations that make them inappropriate to be used in the transaction design process of web applications.

Each ETM, except ACTA (ACTA is a transaction framework), approaches the problem of transaction modeling from a specific point of view. Others try to provide internal structure inside a transaction (Nested) in order to localize failure, while others try to provide isolation relaxation by releasing resources prior to the termination of the entire long lived transaction (SAGAS, Open Nested, ConTract).

It is obvious that the complexity imposed by web applications and the interface diversity of resources cannot be captured by one single model from the above. The main limitation of those models comes from the fact that all members of a transaction structure (sub-transactions and top-level transactions) have the same behavior and semantics. That is, they define once the behavior of the entire structure, and this behavior has to be followed by all the transactions and sub-transactions of the graph. In transactional web applications we need models that are able to accommodate different behavioral patterns into the same structured transaction. For example, a structured transaction that has some of its children to be visible and some others to be invisible.

In addition, in web applications not all activities are strict ACID transactions. Thus we need to model those activities as «weak transactions»; transactions that do not have to satisfy the entire set of ACID constraints. All proposed ETMs provide relaxation of those properties by decomposing a complex transaction into smaller ACID transactions. These models do not give the ability to define for a single flat activity a subset of those properties without decomposition of that activity. Many times in web applications this is a requirement. Consider for example an activity that, as part of a complex transaction, authorizes the user in the system. This authorization activity has to be atomic in order to ensure that all required operations have been successfully executed, but it doesn't need to be durable, or isolated.

The ability of defining «weak transactions» provides a great flexibility in the following circumstances:

- Accommodation of different behavioral patterns into the same structured transaction. We can precisely describe parts of an ACID transaction that are weaker and do not violate the properties of the entire transaction. Very common in web applications.

- Bottom-up transaction design. When integrating legacy systems and diverse resources into the same transaction, it is required to take into account their limitations and built a correct transaction model that respects their behavior and sets proper dependencies between them.

The ACTA framework can be used to analyze and describe transaction models by defining appropriate axioms and using first order logic. We can say that ACTA is closer to describing web transactions, since it can define new models that may be used in an application. However, it has two main limitations:

- **It's too low level for design**. ACTA describes transaction models using first order logic and mathematically expressed axioms to reason about the behaviour of a new model. Although that such a mechanism provides powerful formalism, it is too low level and inappropriate for designing applications. The design process requires high level languages that will be handy and easily understood by implementers, other designers, customers, etc.

- **It cannot describe transactions in a bottom-up fashion**. As with all models that were previously presented, ACTA defines once the behaviour of a complex transaction model and that behaviour has to be respected by all transactions and sub-transactions of the model. It cannot accommodate different behaviours into the same structured transaction.

The above discussion makes clear that there is a need for a high level transaction modelling mechanism for web applications. Such a mechanism should be very flexible in order to allow transaction design in both top-down and bottom-up fashion. It should be also able to describe «weak transactions», and more over, it should provide description of transactions conforming to all presented ETMs when such behavior is needed.

2.2. The Unified Modeling Language

In this section we briefly introduce the Unified Modeling Language [26] in order to familiarize the reader with this language and to enhance his understanding of the UTML.

2.2.1. What is the Unified Modeling Language?

The Unified Modeling Language is a language that unifies the industry's best engineering practices for analyzing and designing software systems. The UML:

- ***Is a language.*** It is not simply a notation for drawing diagrams, but a complete language for capturing knowledge (semantics) about a subject and expressing knowledge (syntax) regarding the subject for the purpose of communication.

- ***Applies to modeling and systems.*** Modeling involves a focus on understanding (knowing) a subject (system) and capturing and being able to communicate this knowledge.

- Is ***the result of unifying*** the information systems and technology ***industry's best engineering practices*** (principles, techniques, methods, and tools).

- Is used for ***specifying, visualizin***g, ***constructing***, and ***documenting*** systems.

- Is used for ***expressing the artifacts*** of a system-intensive process.

- Is based on the ***object-oriented paradigm***.

- Is an evolutionary ***general-purpose, broadly applicable, tool-supported, industry-standardized modeling language***.

- ***Applies to a multitude of different types of systems, domains, and methods or processes.***

- Enables the capturing, communicating, and leveraging of strategic, tactical, and operational knowledge to facilitate ***increasing value*** by ***increasing quality***, ***reducing costs, and reducing time-to-market*** while managing risks and being proactive with respect to ever-increasing change and complexity.

2.2.2. Diagrams in the UML

A *diagram* is the graphical presentation of a set of elements most often rendered as a connected graph of vertices (things) and arcs (relationships). The designer draws diagrams to visualize a system from different perspectives, so a diagram is a projection into a system. For all but the most trivial systems, a diagram represents an elided view of the elements that make up a system. The same element may appear in all diagrams, only a few diagrams (the most common case), or in no diagrams at all (a very rare case). In theory, a diagram may contain any combination of things and relationships. In practice, however, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software-intensive system. For this reason, the UML includes nine such diagrams:

- **Class Diagram.** A *class diagram* shows a set of classes, interfaces, collaborations and their relationships. These diagrams are the most common diagrams found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

- **Object Diagram.** An *object diagram* shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

- **Use Case Diagram.** A *use case diagram* shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

- **Sequence and Collaboration Diagrams.** Both *sequence diagrams* and *collaboration diagrams* are kinds of interaction diagrams. An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages; a collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

- **State-chart Diagram.** A *state-chart diagram* shows a state machine, consisting of states, transitions, events and activities. State-chart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

- **Activity diagram.** An *activity diagram* is a special kind of a state-chart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects. However, UML specification defines that activity diagrams do not provide additional semantics to state-charts. They are used to show the flow of control when system's activities complete.

- **Component diagram.** A *component diagram* shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

- **Deployment diagram.** A *deployment diagram* shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of architecture. They are related to component diagrams in that a node typically encloses one or more components.

This is not a closed list of diagrams. Tools may use the UML to provide other kinds of diagrams, although these nine are by far the most common you will encounter in practice.

2.2.3. Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways. The UML's extensibility mechanisms include:

- **Stereotypes.** A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C_{++}, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first class citizens in your models-meaning that they are treated like basic building blocks-by marking them with an appropriate stereotype.

- **Tagged Values.** A *tagged value* extends the properties of a UML building block, allowing you to create new information in that element's specification. For example, if you are working on a shrink-wrapped product that undergoes many releases over time, you often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block.

- **Constraints.** A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones.

These three extensibility mechanisms allow you to shape and grow the UML to your project's needs. These mechanisms also let the UML adapt to new software technology, such as the likely emergence of more powerful distributed programming languages. You can add new building blocks, modify the specification of existing ones, and even change their semantics. Naturally, it's important that you do so in controlled ways so that through these extensions, you remain true to the UML's purpose-the communication of information.

2.3. Summary

In this chapter we presented the most known Extended Transaction Models. The limitations of these models to be used in complex transactional web applications come mainly from their inflexibility to incorporate different behavioral patterns into the same structured transaction and to accommodate services and resources with diverse transactional semantics and interfaces. Also, they are too low level, (typically first order logic) and thus inappropriate to be used in the design process of web applications.

We also presented the Unified Modeling Language, its main concepts and tools as well as its extensibility mechanism, in order to set up the context into which the UTML notation has been developed.

3. THE TRANSACTION META-MODEL OF UTML

UTML consists of two main parts: the transaction meta-model and the notation system. In this chapter we present the basic concepts of the transaction meta-model, as well as their structure, correlations and dependencies.

A meta-model is a mechanism for describing models. In effect, a transaction meta-model is a mechanism for describing transaction models. The choice of meta-modeling is a necessity originating from the complexity of web transactions. As demonstrated, no specific transaction model can meet the mass of requirements that modern transactional web applications impose. To this end, a transaction meta-model should be both flexible enough to describe the real world and adequately formal to regulate the use of transactional concepts.

Moreover, recall that UTML is a transaction modeling mechanism, and thus it should be able to support all modeling alternatives. In general, designers (regardless what are they designing –application logic, software, system architecture, etc.) desire a rich toolkit (modeling elements and rules) to apply the most appropriate design for each case.

In section 3.1 we present the notion of operation as it is considered in the transaction meta-model. An operation is the minimum slot of work that can be considered in UTML and we distinguish between different types of operations that can be encountered in a transactional web application.

In section 3.2 we present the concept of activity. An activity is a set of operations that are grouped together in order to satisfy a specific user goal. Activities may have to obey specific (formal) execution contracts, and its specification to this direction includes several related concepts.

The different execution contracts that have been defined in the meta-model are formalized by appropriate well-formedness rules and they are presented in section 3.3.

In section 3.4 we define the concept of compensation. A compensation is a special type of activity and is used to semantically undo an executed activity. Operations, activities and compensations are all combined in section 3.5 defining the transaction meta-model that constitutes the basis on which UTML has been built.

Into the following sections of this chapter we present the extensibility mechanism of the meta-model, the well-formedness and well-behaving rules, as well as some guidelines how to define complex transaction closures with UTML.

3.1. Operations

Web applications give the ability to interactive users to invoke certain operations. An operation is the basic functional element that can be encountered in an application. Operations are atomic; either carried out completely or not at all. They can be thought as the minimum slot of functionality that can be provided by an application and thus they cannot be decomposed into sub-operations

Definition 1. *An operation P is a non-suspendable, atomic unit of work that can be executed in the context of an application and it is not further decomposed in the modeling process.*

According to the above definition, the concept of operation, as it is concerned in the meta-model, has two main characteristics:

- ***Non-suspendable***. Operations cannot be suspended and continue their execution later on. Neither the user nor the system can initiate an operation that will cross more than one execution sessions.

- ***Atomic***. Either execute to their completion or not at all. Atomicity of operations has the same meaning with the atomicity of transaction. If an operation cannot complete successfully, then the system (application state or data state) should remain unaffected; as if the operation had never been started.

- ***Indivisible***. Operations cannot be synthesized by other operations. An operation is the smallest piece of work that can be executed in the scope of an application. Of course their logic is not the same for all operations. Indivisibility of operations means that in the modeling process the designer is not interested in the internal structure of an operation or the specific implementation of its logic.

The simplest type of operation that is encountered in any web application is the operation of following hyperlinks. Even simple static web sites give the end users the ability to follow hyperlinks and to navigate through the site. A more complex type of operation concerns the searching functionality that many sites provide. Such operations are triggered by the end user, they access some database (or databases) and they give back to the user a result.

Operations that are invoked by end users are called ***user-triggered*** operations. User-triggered operations are interfaces of the application's functionality to the interactive user. However, user-triggered operations are not the only ones in an application. In this meta-model the following types of operations are identified:

- **Context-triggered operations.** The context of an application is the part of its environment (in which the application operates) that it is interested in. Location aware applications for example are interested for the location of the user. Such a parameter refers to the context of the application. A change in some context parameter may trigger the execution of a certain operation or group of operations. As another example, a disconnection could trigger a "suspend" operation in order to suspend the current executing process or processes that the user had initiated in the application. Regardless of what suspension means.

- **Business logic triggered operations.** The term "business logic" refers to the logic that an application implements in order to enforce the enterprise's business rules. To this end, operations may be defined to implicitly start after some other operations (or as a result of some event) in some order.

Operations have an operation type, and when they are executed, they return results of a specific result type to both the application and the interactive user. The results of a user-triggered operation are in practice a ***data cache*** of a specific type for the application. For example the results of search operation can be of type "SearchResults". All types of results should be explicitly defined in the design process of a specific application. Subsequent operations (possibly data type specific) may be performed on these results. It is obvious that such operations do not change the state of the system and its resources (databases, etc.).

Also, we distinguish between application's ***public data*** (or shared data) and ***user-private data*** that are stored to the user's workspace and can be accessed only by him. In web applications this is a common practice, since many times end users of the application have their private workspace (i.e. shopping cart). However, it is possible other (properly authorized) users to access data stored in the user's private workspace (e.g. sales manager can view the shopping cart of end users). The purpose of this concept is to distinguish between data, on which there is a high competition, and data on which the competition is not low. Thus, in the rest of this document public data refers to application's core data (shared data between many users), while the term private data refers to data that are stored in the private workspace of a specific user (which may be held in the same database with public data, but is dedicated to one user). Data cache should not be confused with private data, since it contains data that are not to be reflected back to the application's resources, whereas private data are held in these resources and can be updated when the user (owner) decides.

Operations on system resources' data may change the state of the application by modifying some of its data. Operations (on cache or resources' data) may or may not change the state of some activities. Typically, operations that are executed in a web client are operations on results that have been generated by other operations in the server and have been transferred to the client, since it is a common practise for some applications to make available part of their functionality on the client's machine.

User-triggered operations and business logic-triggered operations may be ***synchronous*** or ***asynchronous***. An operation is asynchronous when its evaluation does not start immediately and can be done at any given time in the future. Asynchronous operations do not block the user or the system from executing other operations before their evaluation terminates. However, a part of the application's functionality may be disabled due to the fact that an asynchronous operation has not yet been evaluated. We say that asynchronous operations are initially ***submitted for execution*** and after some time they are really evaluated and executed by the system.

Operations are always executed in a ***scope***. In the simplest case this scope is the application's scope. In more structured environments operations are executed in the scope of activities or sub-activities. Thus their scope is the containing activity. The operation concept, as it has been defined in this meta-model is depicted on Figure 7.

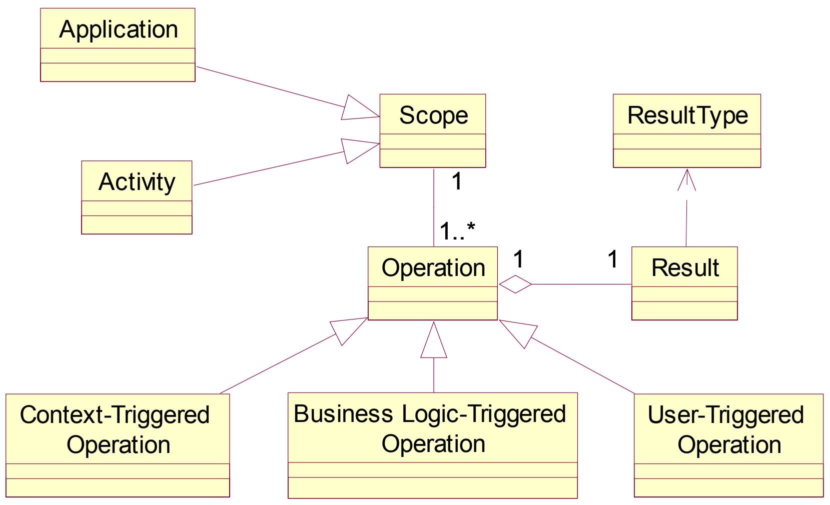


Figure 7: The concept of operation. Clasification and relationships

3.2. Activities

User-triggered operations export the application's functionality to the end user. Usually, such operations are grouped together in order to accomplish complex tasks or to support the achievement of a specific user goal. They may also be combined in many different ways and supplemented with many other operations to satisfy device, environmental and user profile variations. Thus, actually the same application logic results in different ***application views***. For example, the functionality that an application offers to the user through a mobile phone may be (or seems to be) different from the functionality that the same application offers to users that execute the application through their PC (e.g. through your mobile phone you cannot download a file, while through your PC this operation is available). Also, in highly personalised applications a user can choose to access a sub-part of the offered application functionality. The application logic itself imposes several constraints on the order with which operations can be invoked and the design of the application has to take into account those constraints. We use the concept of ***activity*** to describe a set of operations that implement logical parts of the application's functionality which may impose ***constraints*** on the possible operation invocations.

Definition 2. *Activity 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 certain ***signalsets*** appear. The status of an activity can have the following values:

- **Enabled.** All preconditions (e.g. constraints on the execution flow) of the activity are satisfied and it can be started. In the TSS for example, the activity «hotel reservation» is initially disabled. However, after the execution of the activity «user authorization» it gets enabled.

- **Disabled.** Some preconditions of the activity are not satisfied and it cannot be started. In the TSS for example, activities «flight reservation», «hotel reservation» and «event ticket reservation» are disabled until the user gets authorized.

- **Executing.** The activity has been initiated and is currently executing its logic.

- **Executed.** The activity has completed its execution. Completed activities can be enabled again and they can be re-invoked. Executed activities are distinguished into:

    - **Succeeded.** The executed activity has terminated successfully.

- **Failed.** The executed activity has failed and thus it didn't complete successfully. In the TSS for example, if the activity "flight reservation", didn't manage to reserve a ticket, then it is said to have *failed.*

- **Compensated.** The executed activity had succeeded but the system or the user has compensated it (cancel it). This is done by executing another activity. It is possible that not all the effects of the activity have been negated.

- **Suspended.** The activity has been started, executed for a while, and now is not active. It is expected that suspended activities will resume later on and continue their execution. After their resuming, activities become *executing.* Again in the TSS example, the activity hotel reservation can be suspended before the user pays the reservation cost.

An activity can register to a specific signalset for one or more events that it is interested in. A signalset is a set of signals (flags possibly represented by some data) that represent the occurrence of some specific events. Such an event could for example be the termination of an activity, which could be used to enable other activities or to start other activities. When a signal appears in a signalset, a specific action may be taken to change the status of activities that have registered to this Signalset.

Consider for example the case where a user has started and suspended the activity «hotel reservation» in the TSS and left the system. When a signal, denoting a new entrance of the user to the system (the successful termination of the activity «user authorization») appears, an action could be taken to change the status of the activity "hotel reservation" from *suspended* to *executing.* In other words, SignalSets contain signals that can fire specific transitions or trigger special actions, which can change the status of registered activities. Another example of signal is the change in some context parameters. Suppose that an activity is suspended due to a disconnection. The event of re-establishing the connection (change in the value of the context parameter bandwidth) produces a specific signal. This signal can be directed to the suspended activity (through its registration) and trigger its resume operation.

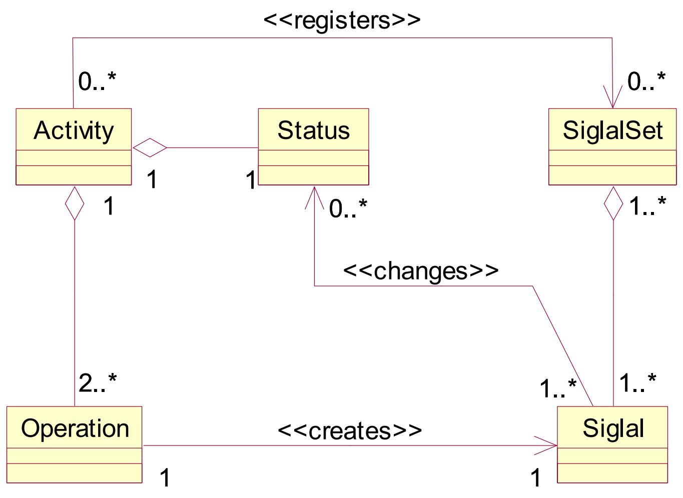


Figure 8: Status Change Model

On Figure 9 the possible status change of an activity is depicted. Each transition is fired when specific signals appear.

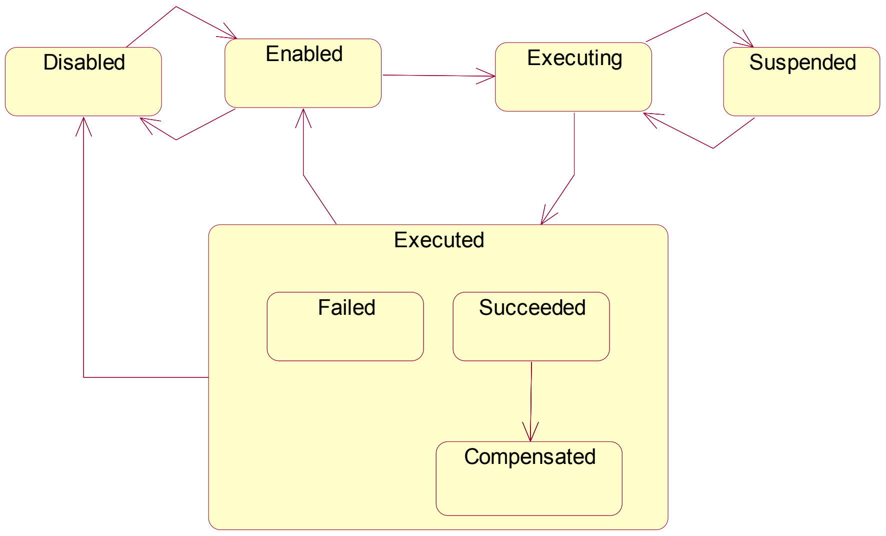


Figure 9: The possible status changes of an activity

An activity may be a ***simple*** (composed of a set of operations), or a ***composite activity,*** which may be composed of other activities and operations. Not all operations within an activity have to be necessarily executed by the user or the system. The same holds for the sub-activities of a composite one. There is a distinction between ***vital*** (obligatory) and ***non-vital*** (optional) operations and sub-activities. Vital operations and sub-activities have to be invoked (by the user or the system) and executed successfully before the activity terminates, although not in a specific order. This does not hold for non-vital ones.

For re-usability reasons, whether a sub-activity is obligatory or optional, it is not a property of the activity itself; rather it is defined by the decomposition semantics attached to the association between parent and sub-activity (the same must hold for activities and operations belonging to their FunctionalSet).

Definition 3. *The Decomposition Association between an activity A and a sub-activity B, DA(A,B), is the semantic* *relationship between the parent and sub activity and is defined on the basis of vitality and visibility of the sub-activity.*

The decomposition association between a parent and sub-activity describes the decomposition semantics. The term decomposition semantics refers to the specific behaviour that parent and sub-activity have during their execution. In particular, this behaviour primarily refers to the termination process of sub-activity. To be clear what exactly definition 3 implies, the precise semantics of visibility and vitality must be provided.

- **Visibility.** It refers to whether a sub-activity makes its results visible to any other activity currently being executed in the system. When a sub-activity is defined to be visible, then when it terminates, its results become visible to any other activity, regardless whether it belongs to the same structure or not. On the other hand, when an activity is defined to be invisible, then if it terminates prior to its parent termination (e.g. sub-transactions in the Nested Transaction model), the only activity that can see its results, is its parent. However, the children (named siblings) of a composite activity are allowed to access objects that have previously been accessed by their parent. Thus, when an invisible activity terminates, its results actually become visible to its parent and siblings.

- **Vitality.** It refers to whether a sub-activity is obligatory or optional when used as a sub-activity of a composite activity. The successful execution of all vital sub-activities is required in order the parent activity to terminate successfully. On the other hand, when a sub-activity is defined as non-vital, then its initiation and execution is up to the user's desire. However, if the user explicitly wants to execute it, then its failure must be reported to him in order to take any possible action and to have a clear picture of what exactly happened. If a non-vital sub-activity is triggered by the system and not the user, then regardless whether it fails or succeeds, the parent activity can terminate successfully.

The question here is why are activities decomposed? There are several reasons to decompose an activity into sub-activities. Some of them are:

- **Failure locality.** By decomposing activities into sub-activities you can localize failures. That is, if an activity fails, it can be re-attempted for "many" times until it succeeds. The number that a sub-activity can be re-attempted is configurable and can be set appropriately by the application designer. Also, some sub-activities may be not so vital for the successful execution of the parent activity. Thus, they can fail (even after they have been re-attempted) without causing the failure of the parent activity.

- **Isolation relaxation.** Many times web transactions take too much time to be completed. In such a case, it's a tragedy for the system's performance to hold recourses locked until the entire, long-lived, transaction terminates. Thus, it is appropriate to release resources as soon as possible. By decomposing a long-lived activity into sub-activities, this can be done as soon as each sub-activity terminates. Such sub-activities are defined to be visible in the sense that they can make their results visible to any other activity currently running in the system and thus, releasing resources prior to the termination of the entire (parent) activity.

- **Resource diversity.** As it has been mentioned, web applications many times access resources that are distributed and have diverse semantics and interfaces. Such resources cannot be accessed with the same approach in managing critical database updates. Thus, many times different transaction processing techniques may be needed. To deal with such situations, the application designer may have to decompose a complex activity (in whose scope diverse resources will be accessed) into smaller sub-activities with appropriate transaction management functionality for each resource.

- **Functionality Integration.** Web applications may be composed of new as well as pre-existing logic, i.e. legacy systems. Consider for example a web application that utilizes the legacy information system of the enterprise and a web service offered by another organization. The transactional properties and semantics of the legacy system and the web service will not, in most cases, be the same. A way to deal with these functionality variations is to decompose a complex activity into smaller activities, each one appropriate to integrate the functionality offered by existing systems or services.

To better handle the operations and sub-activities of an activity, appropriate modeling concepts should be defined. Such concepts are the OperationSet and ActivitySet correspondingly.

Definition 4. *Each activity A has an OperationSet, OS(A), containing all operations that can be invoked in the scope of this activity.*

As mentioned, an operation is considered to be ***obligatory*** (vital) if a successful execution of at least one instance of it is required for the successful termination of the containing activity. Otherwise, it is considered to be ***optional*** (non-vital). As with activities, operations are reusable model elements whenever they are needed during the modelling process of a specific application. Thus, an operation could be defined to be optional when it is used in the scope of a specific activity and obligatory when used in the scope of another activity. The same holds for activities that are used as sub-activities of other complex activities.

Some of an activity's operations implement its logic. An example would be an operation that searches for available seats in a particular flight in the execution of the activity «flight reservation» of the TSS. These operations are called ***functional operations*** and are used to implement the part of application's functionality that is provided through an activity.

Definition 5. *The operations that belong to the OperationSet of an activity A and are used to implement its logic, constitute the FunctionalSet, FS(A), of this activity.*

Some operations of the Activity's OperationSet are used to start, terminate and generally manage the activity's execution or completion, while some others are used to implement the logic of the activity. So, it is useful to distinguish between these types of operations.

Definition 6. *Operations that belong to OperationSet of an Activity A and are used to manage the Activity, compose the ManagementSet, MS(A), of this activity.*

$$OS(A) = MS(A) \cup FS(A) \qquad \text{[Formula 1]}$$

Initially, the meta-model defines that the management set of an activity in UTML can be a subset of the set {«begin», «begin_inv_sub», «begin_vis_sub», «begin_inv_vital_sub», «begin_vis_vital_sub», «end», «commit», «abort», «delegate», «suspend», «resume»}. However, this set is open and can be extended for future use in order to support new models and activity structures. Management operations can be explicitly or implicitly invoked by the user. For example, the user may have the ability to explicitly abort an ongoing activity by pressing an available cancel button. On the other hand, the abort operation can be implicitly invoked when a system failure occurs or some constraints on the activity execution are not guaranteed.

Although the meaning of some of the aforementioned operations is obvious, it would be useful to specify what the precise semantics of each operation are, and what each operation exactly does.

- **begin.** It is used to start a top-level activity. Top-level are the activities that are directly executed in the scope of the application. That is, they are not sub-activities of other activities, called parents.

- **begin_inv_sub.** This operation is used to start an activity, which will be sub-activity of another activity. The decomposition semantics between these activities (parent and sub) define that the child activity cannot commit its operations, but it must delegate the responsibility for that to its parent. This will be done through the ***delegate*** operation.

- **begin_inv_vital_sub.** This operation is a special case of the previous operation and used to start a sub-activity, which is vital in the context that it is used in. That is, its successful execution is required for the successful termination of its parent. If this sub-activity fails then its parent must also fail.

- **begin_vis_sub.** This operation initiates a sub-activity, which since it terminates successfully it can commit the operations for which it is responsible and make its results visible to other activities of any level.

- **begin_vis_vital_sub.** This operation is a specialization of the *begin_vis_sub* operation and it is used to start a sub-activity that its successful termination is required in the context that it is used. Also, this vital sub-activity can commit the operations for which it is responsible prior to the termination of its parent.

- **end.** This operation is used to terminate activities that their completion does not have to satisfy any constraint (except constraints on the execution flow of internal operations). It just changes the status of such an activity from «executing» to «executed».

- **Commit.** It is used to terminate activities that are at least atomic and guarantees that either all constraints that apply on this activity are satisfied and the activity will terminate successfully, or in case that at least one constraint is not satisfied, the activity will abort. In the first case the activity's status changes from «executing» to «succeeded», whereas in the second case the activity's status changes from «executing» to «failed».

- **Abort.** It is used to terminate activities that are at least atomic and it ensures that no partial result of the activity will survive to the system. The status of an activity that terminates with this operation changes from «executing» to «failed». It should be noted that not all the operations of an activity can be cancelled; only operations that make data modifications can be undone. To make it clear, consider that an executed operation reads some data items from a database. What could the rolling back of this operation do? On the other hand, consider an operation that modified the value of a variable, cancelling of this operation means recovery of the initial variable's value.

- **delegate.** The operation *Delegate(OH(b),a)* is used to terminate activities that are at least atomic. It means that the activity *b* gives to activity *a* the responsibility to commit the operations belonging to its *OperationHistory* (definition 10). It takes the place of the operation *Commit* but it checks if all the properties of the activity are guaranteed as the operation commit does. In other words, the commitment or abortion of *b*'s modifications (schedule) is now responsibility of the activity *a*. After its execution, the status of the activity changes from «executing» to «executed». However, it cannot be considered as «succeeded» or «failed», since there is no guarantee whether the actual results of the activity will be committed or aborted. Delegation is important in the case of invisible sub-activities. The term invisible sub-activity is used to describe sub-activities that, when they terminate, make their results available only to their parent and, through it, to their siblings. However, they cannot commit their operations, since their commitment makes the data modifications permanent and visible to all others. Thus, they delegate the responsibility for committing their operations to their parents. On the other hand, the term visible sub-activity is used to describe sub-activities that when they terminate make their results available to any other activity by committing. Related to operation delegate is the operation *responsible(OH(a))* that identifies (returns) the responsible activity for the termination of the OperationHistory (executed operations) of the activity *a*. The invocation of the operation delegate has some constraints that can be summarised as follows:

- An activity A must be the *responsible* activity of an operation P in order to *delegate* the responsibility for committing this operation to another activity.

- The operation *delegate* cannot be invoked after the operation history of an activity has already committed or aborted.

The management set of an activity, as it has been defined, is quite general. The management of an activity concerns its initialization, its termination and in general other operations that handle the execution of the activity regardless its logic or implementation. Each operation of those has its own special semantics and it is used to define a specific behaviour for the activity. Each transaction model is based on special operations, with which transactions can start or terminate, and precise decomposition semantics. Thus, it is important to distinguish between operations that are used to initialize and terminate activities.

Definition 7. *The InitializationSet of an activity A, IS(A), contains the operations that belong to the ManagementSet of the activity and are used to manage its initialization.*

Such operations are: begin, begin_inv_sub, begin_vis_sub, begin_inv_vital_sub, begin_vis_vital_sub, split, etc. This set is open and can be extended for future use in order to support new transaction models. Initialization operations can be explicitly invoked by the user (through an appropriate interface), or implicitly when a functional operation of the activity is requested. For example, the user can explicitly start the activity «BuyBooks» by choosing the appropriate user interface option or the activity can implicitly start when the user executes the operation «AddToCart» for a book that he is interested in. In general, there are no strict requirements for explicit or implicit use of management operations. It is implementation specific and up to the designer’s choice.

In the TSS for example, the activity «flight reservation» could be implicitly started after the user has been authorized and without any other explicit request by him. However, in other implementations the user could have the ability to choose which one of the three activities wishes to execute. In complex activity structures, the initialization operation of a sub-activity can be invoked by the user or its parent activity. In any case, it is considered to belong to the schedule of the activity that it initializes.

Definition 8. *The TerminationSet of an activity A, TS(A), contains the operations that belong to the ManagementSet of the activity and are used to manage its termination.*

Such operations are: commit, abort, end, delegate, join etc. This set is open and can be extended for future use in order to support new transaction models. Each newly defined termination operation must be completely specified along with its semantics and documentation. In general, when defining new management operations one has to explicitly define their documentation, semantics, effects on activity's status, etc.

Definition 9. *The ActivitySet of a composite activity A, AS(A), contains all A's sub-activities. Activities belonging to the same ActivitySet are called Siblings.*

A composite activity's ActivitySet is the mechanism which facilitates to specify its children when decomposing it. It should be stressed that the decomposition semantics between an activity and its sub-activities are not attached to the ActivitySet. This approach is followed by all known ETMs and this is what makes them inappropriate to incorporate different decomposition semantics into the same structured transaction. In this meta-model the decomposition semantics between the parent and each one of its sub-activities is explicitly defined in the decomposition association between them. Thus, we can define different decompositions semantics for different sub-activities in the same structure. This modeling capability is very important, since it provides for accommodating diverse resources and functionality interfaces into the same structured transaction.

The presented concepts and their relationships can be better understood by viewing the diagram in Figure 10. The UML class diagram depicted illustrates the activity concept and its relationship to operations.

To give higher flexibility in the meta-model, activities can be suspended. «Suspend» is a management operation that may be directly or indirectly invoked by the user (for example operations that lead the user outside the activity) or triggered by the context (disconnections). The activity remains suspended until the user returns to continue its execution, or it may be timed out, in which case it is aborted and considered to have failed. Suspending an activity allows the user to navigate and execute other activities (if they are enabled). This is very attractive for the user, who is used to a free navigational environment in the web, but in some cases may create significant system overheads if the designer is not careful. Thus, may appear a need of restricting the set of operations available to the user based on the status of activities in which he participates. This is for the designer to decide, and it is, in general, a trade-off of quality of service (flexibility for the user) versus system performance (overhead incurred).

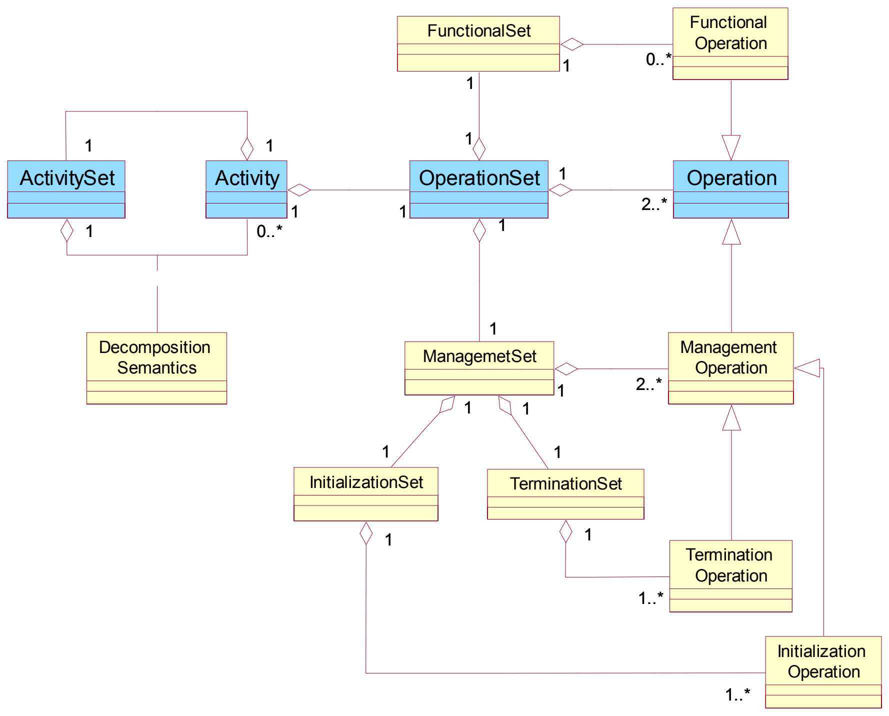


Figure 10: The Activity Concept and its Relationship with Operations

The execution of suspended activities must continue by executing the management operation «resume». «Suspend» and «Resume» operations are management operations that do not belong to InitializationSet or TerminationSet of an Activity. As a consequence, the ManagementSet of an activity A, as it is defined in the meta-model, can be:

$$MS(A) = IS(A) \cup TS(A) \cup \{Suspend, \mathrm{Re}\,sume\}$$ [Formula 2]

In complex structures of activities and operations it is needed to keep track of any executed operation and activity. Of course, a designing mechanism cannot directly model real time behavior based on executed instances of activities. However, it can formalize their behavior by setting appropriate constraints that the application should enforce at run time. To express such constraints some real time concepts concerning the activity execution need to be defined.

For the rest of this thesis we will write $P_i \rightarrow P_j$ to denote the operation P_i precedes operation P_j ordered by their time of execution completion. We will also write {X∘Y∘Z} to denote an ordered set of elements.

Definition 10. *Each activity A has an OperationHistory set, OH(A), that contains all the executed so far operations of A. The OperationHistory of an Activity contains executed operations ordered by their time of execution completion.*

$$\forall P_n \in OH\ (A) \Rightarrow P_n \rightarrow P_{n+1}$$ [Formula 3]

The operation history is created when the initialization operation of an activity is executed and implicitly contains this operation. According to the activity's properties, the OH of an activity can be committed or aborted. The commitment or abortion of the operation history of an activity A, *OH(A)*, implies the commitment or abortion correspondingly of all so far executed operation instances of this activity). For example, an activity which requires that either all its operations are executed successfully or no one is executed, must commit its operation history in order the executed operations to have any effect on the modified data. The operation history of an activity is an ordered set of operations that belong of the operation set of this activity.

An operation history is considered to be complete, when all obligatory operations of the activity have been executed. A complete operation history must be bounded by an initialization and a termination operation instance. As mentioned, any flow of execution defined for the operation set of an activity can be modeled using an *FSM* (Finite State Machine) represented by a state graph. Thus, the OH of the activity is a set of visited nodes, during a traversal of this graph.

However, it is possible that some nodes of the state graph are not included in a complete operation history of a successfully executed activity. The missing nodes represent optional operations of this activity. Also, some operations may appear more than one time in the operation history of an activity. This will occur when an operation is executed more than one times during the activity execution.

Definition 11. *Each composite Activity A has an ActivityHistory set, AH(A) that contains all the so far executed sub-activities of this Activity. The ActivityHistory of an Activity contains the executed sub-activities ordered by the time of their Termination operation execution.*

$\forall X_n \in AH(A) \Rightarrow X_n \to X_{n+1}$ [Formula 4]

As with operation history, an activity history is considered to be complete if all vital sub-activities of a composite activity have been executed. A complete activity history may not mean that the operation history of the composite activity is also complete. It is easily deducted that the execution of the initialization operation of a composite activity precedes the execution of each activity appearing in the activity history of this composite activity. Accordingly, the execution time of the termination operation of the composite activity follows the execution of any activity appearing in its operation history.

It should be stressed that, while the activity history of a composite activity is ordered and for each contained activity its operation history is also ordered, the merged operation history (all executed operations) may be not ordered. In other words, operations of different activities can be intermixed. This happens due to the fact that the order between executed activities is defined by the time of execution completions those activities.

$X, Y \in AH(A) \wedge X \rightarrow Y \not\Rightarrow \forall P_x \in OH(X), P_y \in OH(Y) : P_x \rightarrow P_y$ [Formula 5]

As in case of the operation set of an activity, any possible flow of execution defined for the activity set of a composite activity could be modeled using an FSM and state graphs, which have as nodes sub-activities that belong to the activity set of this composite activity. The activity history of a composite activity is a set of activity instances and represents the visited nodes during a traversal of its state graph. Thus, some nodes (representing non-vital sub-activities) may not be included in the activity history, while others may appear more than once.

Activities and operations may be ***synchronous*** or ***asynchronous***. In the asynchronous case, an operation or activity is submitted to the system for execution after a user's request. The system executes it asynchronously and makes the possible results available to the user a later time. The user may ***join*** the activity to see its results. Operations and activities may also run in ***disconnected mode***. In disconnected mode the client does not communicate with the server and the operations executed in the client are necessarily asynchronous. However, the client may also have copies of some application data. Due to the disconnection, the values of these data in the client machine may differ from the values in the server. In addition, the user may invoke operations that change the values of the copied data that reside in the client. These changes are not visible to the server. When the connection is established again, the client data have to be synchronized with the server data. During this process some of the changes made in the client may not be accepted in the server due to the transactional semantics of execution.

Web Applications are actually activities simply composed of operations or structured in a complex way of other activities (of many types) and ***compensations*** (special activities that are used to semantically undo other executed activities). This meta-model ties together the user interface aspects of free navigation style web applications with structured transactional models that are necessary to accommodate e-business web applications and web services. As any activity, a web application has all the properties and semantics that activities have. A web application has a set of ***web application views*** that share most of the application's logic and are used to export this logic to diverse devices, contexts and user profiles. Note that the activity specification model presented, allows the user to start the execution in one application view and continue it in another. This can be done by suspending an execution activity and continue its execution later on, using the same or other device. The «Plan Trip» activity for example in the TSS can be started using a desktop PC, suspended and then resumed and terminated using a mobile phone.

It should be noted that not all application views are foreseen during the design of an application. Consider for example that initially the TSS was aimed to serve standard PC users. After some years the enterprise owning this application decides to make it available through palmtop devices for mobile users. What the enterprise actually wants is to deliver the same business logic through a different device. A well designed application should be able to re-arrange and re-group its operations into different sets (appropriate for palmtops) and deliver them to the end user. Thus, the same user perceives a different application view when executes the application through his PC and different when it executes it through his palmtop.

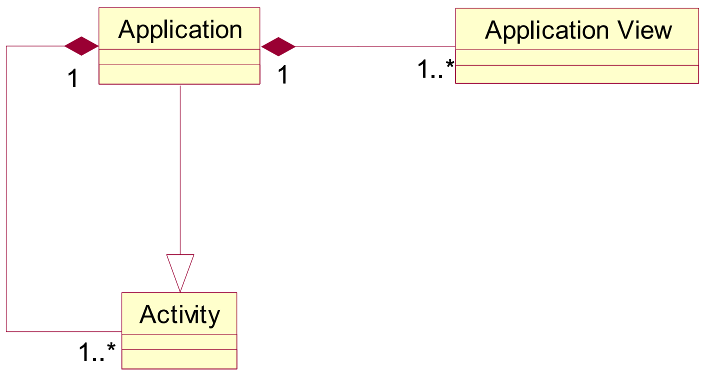


Figure 11: Applications and Application Views

3.3. Execution contracts

Activities may have to obey specific execution contracts. Such a contract defines the constraints that the activity's execution has to satisfy, and actually are used to enforce the business rules of a real world enterprise. In database transactions for example, the execution contract is defined by the ACID properties that a transaction must provide. However, in web applications not all activities have the strict requirements of database transactions. Thus, an activity may have to obey a weaker execution contract than that database transactions have to. An activity's execution contract is defined by using a set of properties that the activity's execution must support. 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 database transaction.

Definition 12. *The PropertySet of an Activity A, PS(A), is the set of properties that the execution of each instance of this activity supports. Such properties are Atomicity, Consistency, Isolation and Durability.*

The meta-model described in this thesis assumes that activities may or may not have to obey an execution contract. If they do, this contract may vary from very strict to very weak. However, the process of defining weaker execution contracts is formalized by appropriate ***well-formedness*** rules. Such rules define formal constraints that must apply on the process of designing web application. The rules that apply on the process of defining execution contracts for activities are described below.

Rule 1.*Activities that support the property «Consistency» must also support the property «Durability».*

$\exists Pr = "Consistency" \in PS(A) \Rightarrow \exists Pr' = "Durability" \in PS(A)$ [Formula 6]

The idea behind rule 1 is that the consistency property implies that any permanent data modification, that an activity makes, must be consistent. It is obvious that an activity must support durability in order to make permanent modification that should be consistent. It is a paradox to say that an activity does not modify any permanent data, and at the same time to say that this activity makes consistent modifications on permanent data.

Rule 2.*Activities that support the property «Consistency» must also support the property «Atomicity».*

$i\exists Pr = "Consistency" \in PS(A) \Rightarrow \exists Pr' = "Atomicity" \in PS(A)$ [Formula 7]

Rule 2 says that since consistency is checked at the termination time of the activity, inconsistent instances of an activity that support consistency, when detected, must be rolled back. This constraint is enforced by defining an activity to be atomic (recall that atomicity requires that no partial effects of an activity survive to the database after its abortion).

Having in mind the rules 1 and 2 (which are actually existence constraints) it is easily proved that the number of the possible different execution contracts that can be defined for an activity, is:

$N = 2^k - 2^{k-2} - 2^{(k-2)-1}$ [Formula 8]

where k, the number of available properties for an execution contract. Thus, the legal execution contracts defined in this meta-model are:

$N=2^4 - 2^2 - 2^1 \Rightarrow N = 10$

The execution contract of an activity A is formed using the initials of each property belonging to its property set and the word «Activity». The execution possible execution contracts for an activity are:

- **No execution contract (empty PropertySet)** $\binom{4}{0}$
    - **Activity.** Activities that have no execution contract to obey. They do not define any additional semantics on their operation or sub-activities. The only constraint that may be defined for them concerns their execution flow. Any activity can define execution flow constraints for its operations and sub-activities regardless whether it has to obey an execution contract or not.

- **Execution Contracts with one property** $\binom{4}{1}$

  - **A_Activity (Atomic Activity).** Activities that have to obey this contract must ensure that either all obligatory operations will be successfully executed, or no one and any successfully executed operation (obligatory or optional) will be undone.

  - **I_Activity (Isolated Activity).** This contract requires that an activity must keep its execution isolated from any other concurrently executed activity. This means that the data that this activity has accessed cannot be accessed by any other activity at the same time.

  - **D_Activity (Durable Activity**). This execution contract defines an activity that some of its operations make data modifications and these modifications have to be durable. Durable means that these modifications survive to the system in any case and cannot be undone by the same activity.

- **Execution Contracts with two properties** $\binom{4}{2}$

  - **AI_Activity (Atomic, Isolated Activity).** An execution contract that includes the all-or-nothing constraint (atomicity) and also supports isolation. That is, activities that have this execution contract must be isolated from any other concurrently executed activity.

- **AD_Activity (Atomic, Durable Activity).** This contract requires that an activity, beyond to all-or-nothing constraint, supports durability for its results.

- **DI_Activity (Durable, Isolated Activity).** It's a contract that forces activities that have to obey it, to make their modifications durable and have its execution isolated form any other concurrently executed activity.

- **Execution Contracts with three properties** $\binom{4}{3}$

  - **ADI_Activity (Atomic, Durable, Isolated Activity).** An execution contract according which, activities must be atomic (all or nothing), durable (permanent modifications) and isolated.

  - **ACD_Activity (Atomic, Consistent, Durable Activity).** This contract implies that an activity must be atomic, durable, and must leave the modified data in a consistent state. If, at the termination of the activity, data consistency constraints are not satisfied, then the activity with be rolled back.

- **Execution Contracts with four properties** $\binom{4}{4}$

  - **ACID_Activity (Atomic, Consistent, Isolated, Durable Activity).** It is the more strict execution contract and the activities that have to obey it, behave like traditional ACID transactions.

The excluded execution contracts due to rules 1 and 2, are the *C_Activity, IC_Activity, AC_Activity, and AIC_Activity, CD_Activity and CID_Activity.*

The sub-typing of these execution contracts (different execution contracts) is depicted on Figure 12.

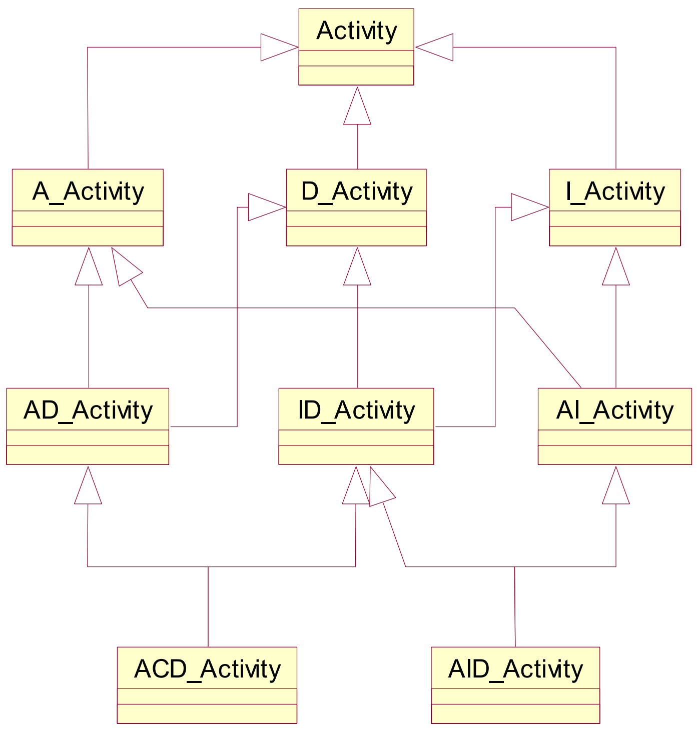


Figure 12: Execution Contract Sub-typing

Using the meta-model described in this thesis the only legal execution contracts that can be defined for activities are those that were previously described. However, in complex activity structures their use depends on the contract of the whole structure and appropriate rules should be defined to formalize their correctness. Such rules are described in section 4.6.1.

3.4. Compensations

An activity may associates with a compensating activity, named compensation. The compensation may be invoked if the user or the system wants to convert an activity that terminated successfully to an aborted activity. However, it should be noted that changing the status of an activity from ***succeeded*** to ***failed*** may have different semantics than changing its status form ***executing*** to ***failed***. A succeeded activity has committed its results and has made them visible to any other activity. Thus, its cancellation cannot simply restore the values that the modified data had before the activity was started. Moreover, sometimes business rules define that activities which have succeeded and are compensated cannot negate all their effects if a certain period of time has elapsed.

Consider for example that in TSS the user has executed successfully the activity «flight reservation» and just three hours before the boarding time decides to cancel his reservation. The airline policy may be to charge the user with a penalty for reserving the ticket for so long. This is a real world example of transaction that is cancelled after having committed. In this case the application has to compensate the activity «flight reservation» by executing the compensation «cancel flight reservation» but it does not credit the user's account with the whole amount that he had paid for the reservation.

Rule 3. *Execution contracts obeyed by compensations must always include atomicity since it should be ensured that all the compensation's defined logic is executed successfully and the results of the compensated activity are successfully rolled back according to the logic defined by the compensation policy.*

$"Atomicity" \subseteq PS(C) \subseteq \{"Atomicity", Consistency", "Isolation", Durability"\}$ [Formula 9]

Definition 13. *Compensation is a special type of atomic activity that is used to semantically undo a successfully terminated activity*

Applying rule 3, the following execution contracts have been defined as legal to be obeyed by compensations:

- **A_Compensation.** It is an atomic activity that is used to semantically undo another activity that has been successfully terminated.
- **AI_Compensation.** It is an atomic compensating activity, whose execution is isolated from any other concurrently executed activity on the same data.
- **AD_Compensation.** It is a compensating activity, whose results are to be permanent after its termination.
- **AID_Compensation.** It is an atomic, isolated compensating activity whose results are stored in permanent storage.
- **ACD_Compensation.** It is a special case of AD_Compensation, which after its execution leaves the database in a consistent state.
- **ACID_Compensation.** It's the more strict execution contract defined for compensating activities and behaves like a traditional database transaction.

The sub-typing of the execution contracts obeyed by compensations is described with a UML class diagram in Figure 13:

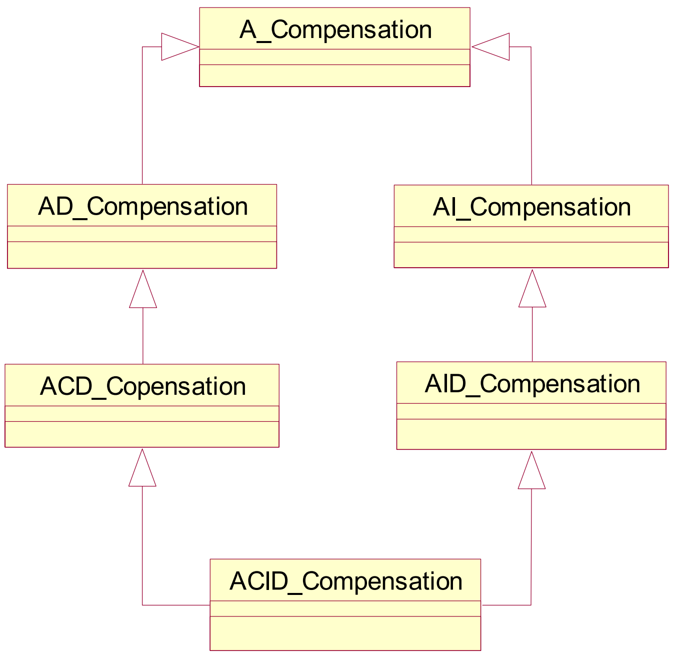


Figure 13: Sub-typing of execution contracts for compensations

As stated, a compensation may not negate all the results of a successfully executed activity. In this case we say that the activity is partially compensated. In this meta-model, partial compensation is supported in two ways. One is that not all activities are associated with compensations. Thus, activities that are not to be compensated in case of failure are not associated with compensations. The other is that compensations can be defined to get input parameters at the point of their initialization. These parameters can influence their behavior. For example, a common parameter to a compensation could be the time elapsed from the termination of the activity that compensates. As described above, this is a common policy for tourism applications when the user changes his mind too late…

3.5. All Together

In this section the entire UTML meta-model is presented. The combination of the aforementioned concepts yields the meta-model depicted on figure 14 as a UML class diagram. In this diagram, operation instances and activity instances are used to represent executed operations and activities correspondingly.

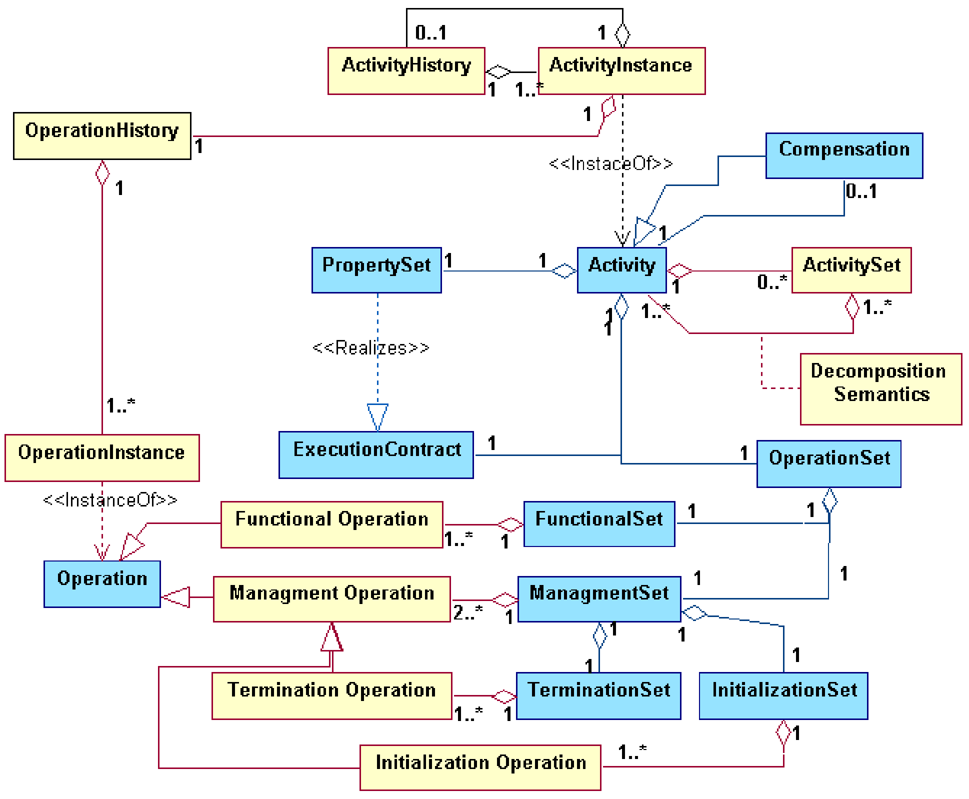


Figure 14: The Entire UTML Meta-Model

3.6. Describing Complex Models

As mentioned, activities can be either simple, or composite. By specifying composite activities, complex structures and hierarchies can be defined in order to satisfy user's, application's or business' logic requirements. Such a structuring can be of any depth. However, during this phase the designer has to be very careful in order to synthesize a correct model. If not, it is possible to assign properties into some activities that conflict with the properties of other activities in different levels of nesting.

In this section formal ***well-formedness*** rules are presented, that can be helpful in using the aforementioned concepts to specify a complex activity model. It is clear that by using appropriate execution contracts and decomposition semantics we can design activity models that are similar to (or more advanced from) well known transaction models, such as Nested Transactions, Open-Nested Transactions, etc. However, the flexibility that this meta-model provides is not boundless.

Also, in order to describe the real time management of activities, appropriate ***well-behaving*** rules are defined. These rules are defined on the basis of real time activity concepts (operation and activity histories) provided by the meta-model.

Each rule (well-formedness or well-behaving) has two parts: The first one describes the rule using an informal natural language (e.g. English), whereas the second describes the rule's semantics in formal mathematical expression.

3.6.1. Well-formedness Rules

Rule 4. *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.*

$$\begin{array}{l} P, P' \quad \textit{properties} \;; A, B \quad \textit{activities} \\ \exists P = \text{"}\textit{Isolation}\text{"} \in PS(A) \Rightarrow \forall B \in AS(A) \exists P' \\ \textit{such} \quad \textit{that } P' \in PS(B) \wedge P' = \text{"}\textit{Isolation}\text{"} \end{array}$$ [Formula 10]

Rule 5. *A composite activity A that has in its ActivitySet at least one sub-activity that supports the property Durability, must also support durability. That is, Durability is upwards transitive.*

$$\begin{array}{l} P, P' \quad \textit{properties} \;; A, B \quad \textit{activities} \\ \exists B \in AS(A) \wedge P = \text{"}\textit{Durability}\text{"} \in PS(B) \Rightarrow \exists P' = \text{"}\textit{Durability}\text{"} \in PS(A) \end{array}$$ [Formula 11]

Rule 6. *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.*

$$\begin{array}{l} P, P' \quad \textit{properties} \;; A, B \quad \textit{activities} \\ \exists P = \text{"}\textit{Consistenc y}\text{"} \in PS(A) \Rightarrow \forall B \in AS(A) \mid DA(A, B).\textit{visibility} = \textit{true} \\ \exists P' = \text{"}\textit{Consistenc y}\text{"} \in PS(B) \end{array}$$ [Formula 12]

Rule 7. *An invisible sub-activity B cannot be further decomposed into visible sub-activities.*

$$\begin{array}{l} A, B, I \quad \textit{activities} \\ DA(A, B).\textit{visibility} = \textit{false} \Rightarrow \forall I \in AS(B) \qquad DA(B, I).\textit{visibility} = \textit{false} \end{array}$$ [Formula 13]

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

$$\begin{array}{l} A, B \quad \textit{activities} \;; O \quad \textit{operation} \;\; P, P' \quad \textit{properties} \\ \exists P = \text{"}\textit{Durability}\text{"} \in PS(A) \Rightarrow \exists O \in FS(A) \\ \textit{or} \exists B \in AS(A) \; \textit{such} \quad \textit{that } \exists P' = \text{"}\textit{Durability}\text{"} \in PS(B) \end{array}$$ [Formula 14]

3.6.2. Well-behaving Rules

Rule 9. *Sub-activities that are defined as invisible must terminate using the operations "Delegate".*

$$A, B : activities; \quad O : operation$$
$$\forall B \in AS(A) \wedge DA(A,B).visible = false \Rightarrow \exists O = \text{"}Delegate\text{"} \in TS(B)$$ [Formula 15]

Rule 10. *A resume operation must follow a suspend operation in the OperationHistory of an Activity.*

$$A : Activity; P, P', P'' : Operations$$
$$\exists P = \text{"}Suspend\text{"} \in OH(A) \wedge P' = \text{"}\mathrm{Re}\,sume\text{"} \in OH(A) \Rightarrow$$
$$P \to P' \wedge \neg(\exists P'' \mid P \to P'' \to P')$$ [Formula 16]

Rule 11. *Activities that are atomic must terminate with one of the operations commit, abort, and delegate.*

$$\mathrm{Pr} : \mathrm{Pr}\,operty; A : activity$$
$$\exists Pr = \text{"}Atomicity\text{"} \in PS(A) \Rightarrow TS(A) = \{commit, abort, delegate\}$$ [Formula 17]

3.7. Choosing Appropriate Compensation Types

When activities are associated with compensations it is not clear what execution contract of compensation is needed for a specific contract of activity. In this section we discuss how the designer can be guided in choosing the appropriate compensation type when he associates activities and compensations.

The execution contract of the successfully terminated activity indicates, in some way, the contract of the compensation that is needed to convert this activity to an aborted one. Rule 3 says that execution contracts that have to be obeyed by compensations must always include the property «atomicity». The following discussion on the other properties that an execution contract may include, will show whether a property included in the activity's execution contract, is required in the execution contract of its associated compensation.

Each property, except "Atomicity", belonging to the PropertySet of an activity is meaningful only in the case that some data are accessed and modified. In particular:

- **Consistency** means that the activity modifies some permanent data and this modification must leave data in a consistent state. In other words, we care about data consistency.

- **Isolation** means that the execution of the activity must be isolated from any other concurrently executed activity. This implies that some of the data that the activity may access are also accessed by other activities and interference may produce inconsistency.

- **Durability** means that at the termination of the activity some of its data modification become permanent and survive future system failures. From a data point of view, durability means that modifications on these data must be reflected in permanent storage.

From the above is clear that the execution contract of an activity is mainly chosen by the quality of data, which are to be modified by this activity. Thus, the compensation for this activity (which operates on the same data) must respect the data quality in the same way as the activity did.

Rule 12. *If an activity A has an associated compensation C(A), then C(A) must contain all properties supported by A plus the Atomicity property.*

$$A \quad activity$$
$$PS(C(A)) = PS(A) \cup \{Atomicity\}$$ [Formula 18]

Table 1 shows the appropriate compensation for each execution contract obeyed by an activity.

Activity Type	Compensation Type
Activity	A_Compensation
A_Activity	A_Compensation
I_Activity	AI_Compensation
D_Activity	AD_Compensation
AI_Activity	AI_Compensation
AD_Activity	AD_Compensation
DI_Activity	ADI_Compensation
AID_Activity	ADI_Compensation
ACD_Activity	ADC_Compensation
ACID_Activity	ACID_Compensation

Table 1: Correspondence of compensation and activities

3.8. The Extensibility Mechanism

The meta-model presented in the previous sections can be extended to support new activity models for specific application domains and requirements. The extensibility mechanism of the meta-model has two parts. The fist part includes extension of the management set of activities. Recall that the management set contains operations that are used to manage the activity execution. The meta-model comes with the following predefined management operations: {begin, begin_vis_sub, begin_inv_sub, begin_inv_sub, begin_inv_vital_sub, commit, abort, end, delegate, resume, suspend}. However, the designer can provide specification of new management operations that will be used by the model that he wants to describe. In order to define a new management operation, the designer has to provide its name and to specify its precise semantics.

The second part of the extensibility mechanism is the well-formedness and well behaving rules. Well-formedness rules provide the basis on which the activity structuring will be done, according to the newly defined model, whereas the well-behaving rules formalize the behaviour of the newly defined model.

3.8.1. An Example of the Meta-model's Extensibility

As an example of the meta-model's extensibility, in this section we extend it to describe a multi-database transaction model. This model is followed by applications that span multiple, remote and heterogeneous database systems. A multi-database transaction (a global transaction) is a collection of sub-transactions executed at local database systems. In addition to the specification of sub-transactions, the designer can specify execution dependencies between them. The sub-transactions of a global transaction are submitted for execution to local database systems. At the time of commitment the global transaction asks its children to prepare for commitment. If every one of its children replies ok, then it asks them again to commit their operations. Thus, we have a bottom-up process of commitment for the global transaction. In case that at least one of the children answers that it cannot commit successfully, the global transaction asks each one of his children to abort. For such an activity model we have the following formalism:

```
Let GA be a global activity, CA be a child activity.
```

1) $IS(GA) = \{begin\}$
2) $TS(GA) = \{abort, commit\}$
3) $MS(GA) = IS(GA) \cup TS(GA) \cup \{suspend, resume, ask\_prepare\}$
4) $IS(CA) = \{begin\_inv\_sub, begin\_inv\_vital\_sub\}$
5) $TS(CA) = \{abort, commit\}$
6) $MS(CA) = IS(CA) \cup TS(CA) \cup \{suspend, resume, reply\_commit, reply\_abort\}$

Semantic specification of new management operations:

- **ask_prepare:** This operation is used from a global activity to ask its children for commit preparation. According to the answers received this activity can abort or commit. In particular, if every child answers that it can commit, then the global activity asks its children to commit. Otherwise it asks them to abort and it aborts all its operations.

- **prepare:** This operation is executed by child activities to examine whether they can commit. If so, they reply accordingly to their parent, without committing. Otherwise, the abort (using the abort operation) and answer accordingly to their parent.

- **reply_commit:** This operation is used by child activities (sub-activities) to send an answer to their parent which says that all are ok and they can commit. If the parent activity (global activity) receives reply_commit from every child activity then it can commit. After the execution of this operation the child activity guarantees that if it will be asked to commit it will do that in any case.

- **reply_abort:** This operation is used by child activities to send an answer to their parent which says that they have aborted during the prepare phase (the execution of prepare operation).

Well-Behaving Rules:

Rule 13. *A Global activity A can commit if all the activities belonging to its ActivitySet answer "reply_commit".*

$$A, A' : Activities$$
$$\exists A' \in AS(A) \vee \exists P = "reply\_abort" \in OH(A') \Rightarrow TS(A) = \{abort\}$$ [Formula 19]

Rule 14. *If an activity A has executed its prepare operation, then it cannot execute any other functional operation.*

$$\exists P = "prepare" \in OH(A) \Rightarrow \forall P' \in OH(A) \mid P \to P' \; then \quad P' \notin FS(A)$$ [Formula 20]

Rule 15. *If an activity A executes a reply_commit operation, then its next operation that will be successfully executed must indispensably be the commit operation.*

$$\exists P = "reply\_commit" \in OH(A) \Rightarrow \exists P' = "commit : P \to P' \, in \quad OH(A)$$ [Formula 21]

Well-Behaving Rules:

Rule 16. *All children of a global activity must be invisible.*

$$GA, CA : Activities$$
$$CA \in AS(GA) \Rightarrow DA(GA, CA).visible = false$$ [Formula 22]

3.9. Summary

In this chapter we presented the transaction meta-model that we propose for modeling complex web transactions. The meta-model proposes the use of operations, activities and execution contracts in order to define units of work that have transactional semantics.

It provides advanced transaction modeling by defining the decomposition semantics for each sub-transaction and gives the ability of defining transactions that accommodate different behavioral patterns into the same structure. It also models «weak transactions» either self-contained or as part of more complex structures. The meta-model provides a rich set of well-formedness and well-behaving rules that are used to formalize the transaction design process and they constitute a correctness criterion to evaluate the produced transaction models.

Moreover, the meta-model itself is extensible and can be extended to meet requirements of specific applications. The extensibility mechanism has two parts. The first one is the management operations used by the meta-model, while the second concerns the well-formedness and well-behaving rules. Such rules can be defined by the designer in order to describe specific behavior which may not be currently captured by the meta-model.

4. THE NOTATION SYSTEM

In this chapter we present the notation system of UTML. The notation system, the well-formedness and well-behaving rules constitute the toolbox of UTML. A notation system for a design language must satisfy specific goals. The goals for the notation system of UTML are:

- **Simplicity.** The notation that a designer would like to use in designing applications must be as simple as possible. That is, a notation system should not impose additional complexity to the application of the theory behind a design language and if possible to reduce it. This is a goal that should be met by the notation system of UTML.

- **Familiarity.** Although that a novel notation system is many times impressive (due to its innovation), its acceptance and utilization by designers rarely gets wide. Thus, another goal that UTML's notation system should meet is to be as much familiar as possible.

To this end, UML has been chosen as the basis on which UTML is built. UML is a widely accepted and used industrial modeling standard that is extensible. Thus, UTML extends it in order to provide the meta-model's functionality through a concrete, handy, and simple modeling tool. Application designers will use this (UML compatible) notation to describe the application's logic and transactional semantics.

We firstly present some advanced UML concepts n section 4.1 that are not so known and they are used in the definition of the UTML notation system. Then, we present the UTML profile structure in section 4.2, as well as the organization and execution model elements in sections 4.3 and 4.4 correspondingly.

CHAPTER 4. THE NOTATION SYSTEM

4.1. Some Advanced UML Concepts

The following UML terms are not widely known, and thus we discuss them before presenting the notation system for convenience of the reader.

- **UML profile:** A UML profile is a stereotyped package that contains model elements which have been customized for a specific domain or purpose, by extending the UML meta-model using appropriate stereotypes, tagged definitions, and constraints. A profile may specify the model libraries on which it depends and the meta-model subset that it extends.

- **Tag Definition:** Tag definitions specify new kinds of properties that may be attached to newly defined model elements. In tag definitions you specify the semantics of the new tags, the stereotype in which they are attached, the multiplicity and their type.

- **Model:** A model captures a view of a physical system. Hence, it is an abstraction of the physical system with a certain purpose; for example, to describe behavioral aspects of the physical system to a certain category of stakeholders. A model contains all the model elements needed to completely represent a physical system, according to the purpose of this particular model. The model elements in a model are organized into a package/subsystem hierarchy, where the top-most package/subsystem represents the boundary of the physical system. Different models of the same physical system show different aspects of the system. The pre-defined stereotype «SystemModel» can be applied to a model containing the entire set of models for a physical system.

- **Composite State:** A composite state is a state that contains other state vertices (states, pseudo-states, etc.). The association between the composite and the contained vertices is a composition association. Hence, a state vertex can be a part of at most one composite state. Any state enclosed within a composite state is called a *sub-state* it. It is called a *direct sub-state* when it is not contained by any other state; otherwise, it is referred to as a *transitively nested sub-stat*e. In UML, composite state is a subtype of state. Attributes:

    - ***isConcurrent:*** a Boolean value that specifies the decomposition semantics between a composite state and its sub-states. If this attribute is true, then the composite state is decomposed directly into two or more orthogonal conjunctive components called *regions* (usually associated with concurrent execution). If this attribute is false, then there are no direct orthogonal components in the composite.

    - ***isRegion:*** a derived Boolean value that indicates whether a composite state is a sub-state of a concurrent state. If it is true, then this composite state is a direct sub-state of a concurrent state.

    - **DeepHistory:** is used as a shorthand notation that represents the most recent active configuration of the composite state that directly contains this pseudo-state; that is, the state configuration that was active when the composite state was last exited. A composite state can have at most one deep history vertex. A transition may originate from the history connector to the *default* deep history state. This transition is taken in case the composite state had never been active before.

- **ShallowHistory:** is a shorthand notation that represents the most recent active sub-state of its containing state (but *not* the sub-states of that sub-state). A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active sub-state of a state. A transition may originate from the history connector to the *initial* shallow history state. This transition is taken in case the composite state had never been active before.

4.2. The UTML Profile

The UML profile that has been defined to provide the UTML notation consists of two main parts:

- **The Organization Model.** It describes the application under design from a static point of view. That is, it conceptually represents the organization of the activities that implement the application, their management, their decomposition semantics, and their dependencies. To do so, it utilizes UML class diagrams produced by stereotyped classes and associations. In designing the static structure of the activities that an application utilizes, all well-formedness and well-behaving rules are applied to formalize the modeling process.

- **The Execution Model.** It complements the transaction meta-model that was described in chapter 4 by providing modeling of the execution flow for the application and its activities. Execution model uses state machines and state charts to describe possible execution flow of activities, setting this way real time dependencies between them. Also, by providing execution flow of activities, the application designer can describe a primitive ***user navigation model***. Such a navigation model, sets appropriate constraints on the navigational abilities that the user may have when executing an application. It should be noted that the execution model (state machines) is defined for an abstract state named «application». This is needed due to the fact that UML defines that state machines and state charts can be defined for classes or use cases. This class represents the application under design and the state charts defined in the execution model are actually modeling the application's state (flow of control).

The UTML profile has the structure that is depicted on figure X.

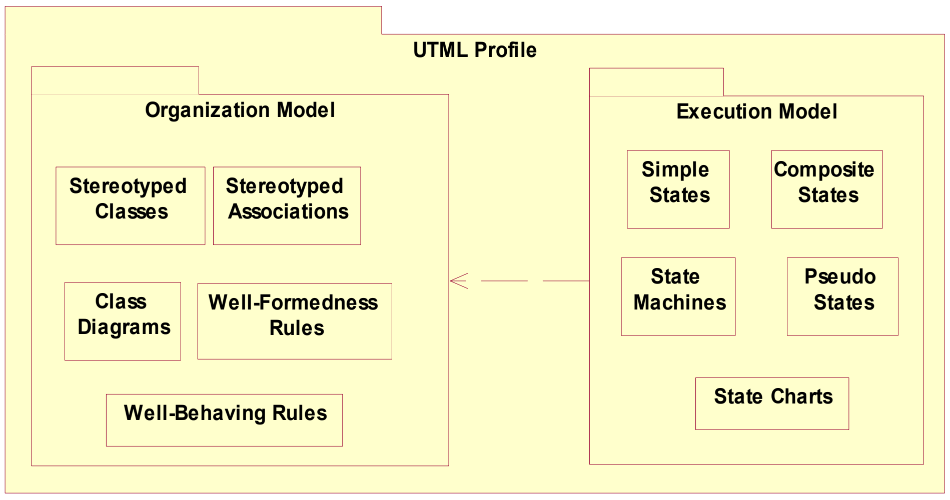


Figure 15: UTML Notation Structure

4.3. Organization Model Elements

In this section the model elements (stereotypes) that have been defined for the organization model are presented. Table 2 shows the definition of all tag values that are used by the stereotypes the following tables shows the definition of the organization model stereotypes.

Tag	Stereotype	Type	Mult/ty	Description
Name	«Activity»	UML::Datatypes::String	1	It is the name of the Activity.
TriggeredBy	«Activity»	TransactionProfile::TriggerEnum (An enumeration: {User,App.Logic,Context})	1	Indicates, who triggers the activity.
IsSimple	«Activity»	UML::Datatypes::Boolean	1	Indicates whether the activity has sub-activities or not
IsStrict	«Activity»	UML::Datatypes::Boolean	1	Indicates whether during the execution of this activity the user has free navigation capabilities in the user interface. Set to true is meaningful only if the activity incorporates user intervention.
IsSynchronous	«Activity»	UML::Datatypes::Boolean	1	Indicates whether the activity is executed asynchronously or not.
TimeOut	«Activity»	UML::Datatypes::UnlimitedInteger	1	Indicates the maximum period of time (in seconds) that an activity can be active. This parameter is meaningful for activities that involve user intervention.
Documentation	«Activity»	UML::Datatypes::String	1	Describes the activity.
mSet	«Activity»	TransactionProfile::ManagementEnum (An enumeration: {suspend, resume, ask_prepare, reply_commit, reply_abort})	1	Indicates the ManagementSet of the activity. ManagementSet contains management operations that belong neither to activity's IS nor to activity's TS
iSet	«Activity»	TransactionProfile::InitializationEnum (An enumeration: {begin,begin_vis_sub, begin_inv_sub, begin_vis_vital_sub, begin_inv_vital_sub})		Indicates the InitializationSet of the activity. InitializationSet contains management operations that are used to initialize an activity.
tSet	«Activity»	TransactionProfile::TerminationEnum (An enumeration: {end,commit,abort,delegate}		Indicates the TerminationSet of the activity. TerminationSet contains management operations that are used to terminate the activity.

Table 2: Tag Value Definitions

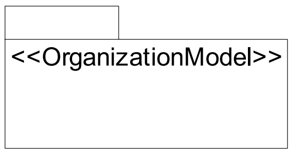
Stereotype	**Organization Model** «OrganizationModel»
Base Class	Model
Parent	Not Available
Description	An organization model is a UML model that describes the organization of the application's activities and their semantic relationships. It describes the application from a static point of view, showing only the organization of transactions without any possible execution sequence. To do that, it uses activities, compensations (of any execution contract) and decomposition associations forming class diagrams.
Constraints	None
Notation	

Table 3: OrganizationModel Stereotype Definition

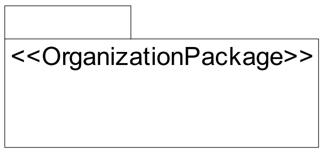
Stereotype	**OrganizationPackage** «OrganizationPackage»
Base Class	Package
Parent	Not Available
Description	An organization package is a package that contains stereotyped classes and associations used in the Organization model. Packages group model elements that are used to describe a specific part of the application.
Constraints	None
Notation	

Table 4: OrganizationPackage Stereotype Definition

Stereotype	**Activity** «Activity»
Base Class	Class
Parent	Not Available
Description	An «Activity» is a class that conceptually represents a unit of work. This work must be done by the system or the user, or both. An Activity is a logical part of application's functionality. The web application that is modeled using this profile is performing an activity at any time. Conceptually, when the application is active, then it must be in the scope of some action.

Tag Values	`Name`
	`TriggeredBy`
	`IsSimple`
	`IsStrict`
	`IsSynchronous`
	`TimeOut`
	`mSet`
	`iSet`
	`tSet`
	`Documentation`

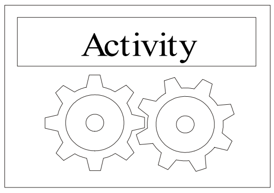
Constraints	None
Notation	

Table 5: Activity Stereotype Definition

Stereotype	**Atomic Activity** «A_Activity»
Base Class	Class
Parent	«Activity»
Description	An Atomic Activity is a specialization of Activity, which has atomicity semantics. That is, it requires that either all its vital operations/sub-activities will be successfully executed,

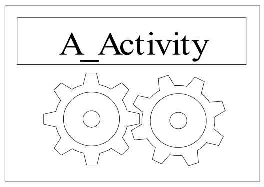
	or no one will be, and any partial result is rolled back.
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 6: A_Activity Stereotype Definition

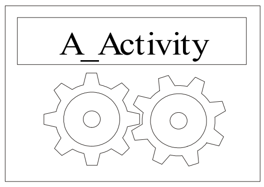
Stereotype	**Isolated Activity** «I_Activity»
Base Class	Class
Parent	«Activity»
Description	An Isolated Activity represents a unit of work that its execution does not interfere with any other concurrently executed activity.
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 7: I_Activity Stereotype Definition

Stereotype	**Durable Activity** «D_Activity»
Base Class	Class
Parent	«Activity»
Description	A Durable Activity makes data modifications that are to be permanent after its termination.
Tag Values	Inherited by its parent
Constraints	None

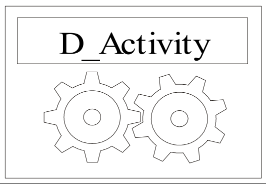
Notation	

Table 8: D_Activity Stereotype Definition

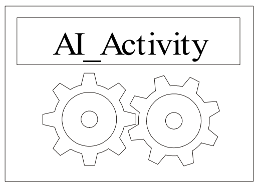
Stereotype	**Atomic Isolated Activity** «AI_Activity»
Base Class	Class
Parent	«A_Activity», «I_Activity»
Description	An AI Activity is isolated while being executing and supports the all or nothing property.
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 9: AI_Activity Stereotype Definition

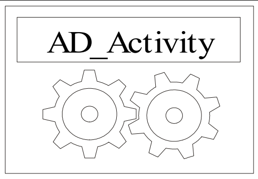
Stereotype	**Atomic Durable Activity** «AD_Activity»
Base Class	Class
Parent	«A_Activity», «D_Activity»
Description	An AD Activity is a specialization of Atomic Activity and its results survive to the system after its termination.
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 10: AD_Stereotype Definition

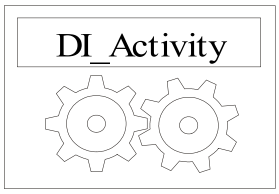
Stereotype	**Durable Isolated Activity** «DI_Activity»
Base Class	Class
Parent	«I_Activity», «D_Activity»
Description	A DI Activity is a specialization of Isolated Activity and Durable Activity. Such an activity makes data modification in isolation and these modifications will survive to the system after the activity's termination.
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 11: DI_Activity Stereotype Definition

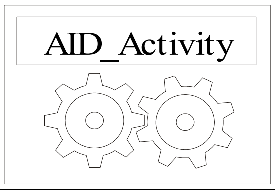
Stereotype	**Atomic Isolated Durable Activity** «AID_Activity»
Base Class	Class
Parent	«AI_Activity», «AD_Activity»
Description	An AID Activity is a specialization of AD_Activity and AI_Activity, composing a new activity type, which has permanent results, is isolated and supports the all or nothing property in its execution.
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 12: AID_Activity Stereotype Definition

Stereotype	**Atomic Consistent Durable Activity** «ACD_Activity»
Base Class	Class

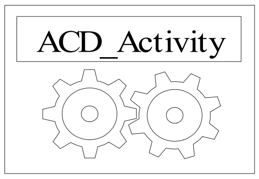
Parent	«AD_Activity»
Description	ACD activities either execute to completion or not at all. Some of their data modifications survive to the system and at the termination of the activity, all integrity constraints are satisfied.
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 13: ACD_Activity Stereotype Definition

Stereotype	**Atomic Consistent Isolated Durable Activity** «ACID_Activity»
Base Class	Class
Parent	«AID_Activity», «ACD_Activity»
Description	An ACID Activity is a derived activity type, which inherits form ACD_Activity and AID_Activity, forming the most strict activity type. Its execution is isolated from any other concurrently executed activity on the same data and it leaves data in consistent state. It also requires that all its operations will be executed successfully and their modification will be permanent even in case of system failure. Such Activities behave like traditional database transactions.
Tag Values	Inherited by its parent
Constraints	None

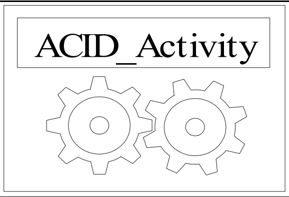
Notation	

Table 14: ACID_Activity Stereotype Definition

Stereotype	**Compensation** «Compensation»
Base Class	Class
Parent	«A_Activity»
Description	Compensation is a group of operations that semantically undo the results of a successfully terminated activity. Compensations may be simple or composite containing other sub-compensations. Compensations are also atomic. That is, either all its operations/sub-compensations are successfully executed or not at all. They have no optional operations and are used to convert a committed activity to an aborted one.

Tag Values	`Name`
	`DriggeredBy`
	`IsSimple`
	`IsSynchronous`
	`MSet`
	`iSet`
	`tSet`
	`Documentation`

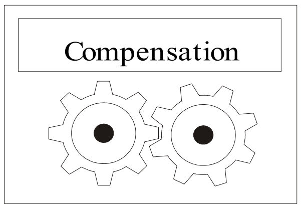
Constraints	Atomicity must be included in its PropertySet
Notation	

Table 15: Compensation Stereotype Definition

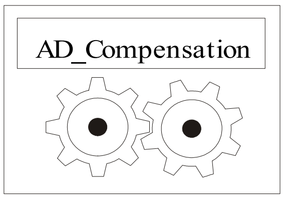
Stereotype	**Atomic Durable Compensation** «AD_Compensation»
Base Class	Class
Parent	«Compensation»
Description	Similar to AD_Activity
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 16: AD_Compensation Stereotype Definition

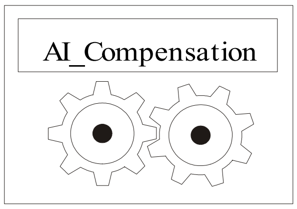
Stereotype	**Atomic Isolated Compensation** «AI_Compensation»
Base Class	Class
Parent	«Compensation»
Description	Similar to AI_Activity
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 17: AI_Compensation Stereotype Definition

Stereotype	**Atomic Isolated Durable Compensation** «AID_Compensation»
Base Class	Class
Parent	«AI_Compensation», «AD_Compensation»
Description	Similar to AID_Activity
Tag Values	Inherited by its parent
Constraints	None

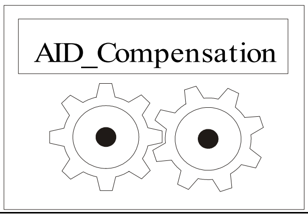
Notation	

Table 18: AID_Compensation Stereotype Definition

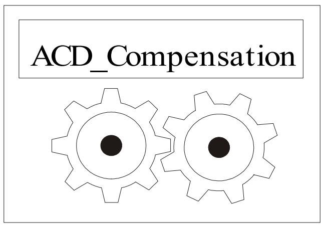
Stereotype	**Atomic Consistent Durable Compensation** «ACD_Compensation»
Base Class	Class
Parent	«AD_Compensation»
Description	Similar to ACD_Activity
Tag Values	Inherited by its parent
Constraints	None
Notation	

Table 19: ACD_Compensation Stereotype Definition

Stereotype	**Atomic Consistent Isolated Durable Compensation** «ACID_Compensation»
Base Class	Class
Parent	«ACD_Compensation», «AID_Compensation»
Description	Similar to ACID_Activity
Tag Values	Inherited by its parent
Constraints	None

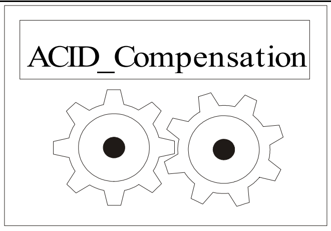
Notation	

Table 20: ACID_Compensation Stereotype Definition

Stereotype	**Compensates** «compensates»
Base Class	Association
Parent	Not Available
Description	Associates an activity (or any specialization) with a compensation (or any specialization). It shows which compensation will be used to compensate an activity instance if it is needed. Rule 12 applies on associations between activities and compensations in order to guarantee that the resulting structure is correct.
Tag Values	None
Constraints	None
Notation	An association line stereotyped as «compensates»

Table 21: Compensates Stereotyped Definition

Stereotype	**Invisible Sub-activity** «invisible»
Base Class	Association
Parent	Not Available
Description	This association connects two activities with parent – sub-activity semantics. It also means that the sub-activity is not vital. That is, if the sub-activity aborts the parent activity can continue its execution and terminate successfully. Moreover, the parent activity follows the Nested Transaction Model, which means that the commitment of the sub-activity depends on the commitment of the top-

	level activity (containing activity) and modifications of sub-activity are made visible to others if and only if the parent activity commits.
Tag Values	None
Constraints	None
Notation	An association line stereotyped as «invisible»

Table 22: Invisible Stereotype Definition

Stereotype	**Vital Invisible Sub-activity** «vital_invisible»
Base Class	Association
Parent	«invisible»
Description	This association is a special case of the invisible sub-activity association. The difference is that the sub-activity is a vital one. That is, if the sub-activity aborts then, the parent activity must also abort.
Tag Values	None
Constraints	None
Notation	An association line stereotyped as «vital_invisible»

Table 23: Vital_Invisible Stereotype Definition

Stereotype	**Visible Sub Activity** « visible»
Base Class	Association
Parent	Not Available
Description	This is a decomposition association between a parent and a sub activity, according to which the sub-activity makes its results visible to others before the parent activity commits. That is, the commitment of the sub-activity is independent from the commitment of the containing activity. If later the parent activity aborts, the execution of an appropriate compensation is needed to semantically undo the results of committed sub-activities. It is clear that the transaction

	model between containing activities and sub-activities that are associated with a «visible» association is the Open Nested Transaction Model.
Tag Values	None
Constraints	None
Notation	An association line stereotyped as «vital_invisible»

Table 24: Visible Stereotype Definition

Stereotype	**Vital Visible Sub Activity** «vital_visible»
Base Class	Association
Parent	Not Available
Description	This association is a special case of the «visible» association. The difference is that the sub-activity is a vital one. That is, if the sub-activity aborts then, the containing activity must also abort.
Tag Values	None
Constraints	None
Notation	An association line stereotyped as «vital_visible»

Table 25: Visible Stereotype Definition

4.4. Execution Model

The execution model utilizes UML state machines and state charts to provide modeling of possible execution flows for an application's activities. While the organization modeling of an application is mandatory, the execution modeling is not required and the designer may or may not provide it.

In execution modeling, all UML model elements that concern the state modeling are available. However, some new model elements have been defined in order to properly support transaction design with UTML.

Stereotype	**Execution Model** «ExecutionModel»
Base Class	Model
Parent	Not Available
Description	An execution model is a UML model that describes the possible sequences of execution for the application's transactions. It describes the dynamic behaviour of the application, showing the possible transitions from an activity to other activities. To do that, it utilizes states, composite states, complex states, transitions, pseudo-states, forks, joins, junctions, etc. forming state charts that represent the execution flow of the application. An execution model describes: • The activities' structure along with execution dependencies between them. • The application's flow of control. • Primitive user navigational patterns
Tag Values	None
Constraints	None
Notation	The notation used for an Execution Model is a model symbol stereotyped as «ExecutionModel»

Table 26: ExecutionModel Stereotype Definition

Stereotype	**Execution Package** «ExecutionPackage»
Base Class	Package
Parent	Not Available
Description	An execution package is a UML package that contains

	states, transitions and other model elements to describe the dynamic behavior of activity models.
Tag Values	None
Constraints	None
Notation	The notation used for an Execution Package is a package symbol stereotyped as «ExecutionPackage».

Table 27: ExecutionPackage Stereotype Definition

Stereotype	**Explicit Start** «ExplicitStart»
Base Class	Pseudo-state-kind
Parent	Not Available
Description	An Explicit Start pseudo-state is used in composite states that include states which represent non-vital, user-triggered activities. It actually provides the user with two operations: one to explicitly start the execution of the optional activity, and one to explicitly bypass the activity. ExplicitStart has at most one incoming transition emanating form an initial state or a history (shallow or deep) pseudo-state vertex, and exactly two outgoing transitions: one leading to the default sub-state and one leading to the final state of the composite state. The first transition is fired when the user explicitly starts the execution of the optional activity, while the second when the user explicitly bypasses the optional activity. This pseudo-state is used to make clear that, non-vital (user-triggered) activities that are modeled as regions inside

	concurrent composite states must be explicitly executed by a user call. In other words, in a concurrent composite state, a non-vital sub-state will be entered, but it is user-dependent if the represented activity will be actually executed.
Tag Values	None
Constraints	None
Notation	A state symbol stereotyped as «EXPLICITSTART»

Table 28: ExplicitStart Stereotype Definition

Stereotype	**Rollback** «ROLLBACK»
Base Class	Pseudo-state-kind
Parent	Not Available
Description	A Rollback pseudo-state is used inside composite states that represent activities defined as atomic. When used, it represents an activity, during which any partial execution of the enclosing activity is rolled back. The logic of this activity cannot be statically modelled, since it depends on run-time information about what was previously executed. Each vital sub-activity of the enclosing activity has a transition leading to this vertex. This transition is fired when the activity (represented by the source state) fails. What actually happens during this state can be described with the following algorithm: 1 Undo all operations belonging to the OH of the activity represented by the enclosing state. 2 Force each suspended sub-activity of the same level to resume and immediately rollback.

	3 Create an appropriate Compensation (using rule 12), named RB, with empty functional set. Select the corresponding compensation (if exists) for each activity belonging to the AH (except those that have terminated using the delegate operation) of the activity represented by the enclosing state and insert it to the AS of RB. Execute RB. 4 If the enclosing state is a region and the represented activity is vital, then: 4.1 Force each same level region, that its ExplicitStart is active, to reach its own rollback pseudo-state immediately. 4.2 Force each same level region, that its ExplicitStart is inactive, to fire the transition leading to its final state 5 Fire the outgoing transition to the final state of the enclosing state with a signal of fail. 6 End When a composite state is decomposed into two or more concurrent sub-states (regions), a rollback pseudo-state for the enclosing composite state is implied to be a direct sub-state of it.
Tag Values	None
Constraints	None
Notation	A state symbol stereotyped as «ROLLBACK»

Table 29: RollBack Stereotype Definition

Stereotype	**Commit** «COMMIT»
Base Class	Pseudo-state-kind
Parent	Not Available
Description	A Commit pseudo-state is used inside states that represent activities defined as atomic. It represents a state during which all executed operations and invisible sub-activities of

	the activity (represented by the enclosing state) take effect. What exactly "take effect" means, depends on the execution contract that this activity has. For example if the enclosing activity is defined as durable, then some of its data modifications will be permanent. A commit state is used just before the final state of the enclosing activity. This is done due to the fact that the committing of an activity is a termination operation. Recall that in this meta-model we are not interested about the internal decomposition of operations. This is what we define Commit as a pseudo-state. The logic of this operation can be described with the following algorithm. 1. `Ask all activities belonging to the AH of the activity represented by the enclosing state and have in its TS the operation` *`prepare`*`, to` *`prepare`*`.` 2. `If all activities answer "reply_commit", then ask them to` *`commit`*`. Else fire transition leading to the ROLLBACK pseudo-state.` 3. `Commit all operations belonging to the OH of the activity`

	`represented by the enclosing state. If commitment was not successful fire the transition to the ROLLBACK pseudo-state. Else end.` If an atomic activity, represented by a composite state, is decomposed in two concurrent composite states (regions) then the commit pseudo-state is implied to be a direct sub-state of the enclosing state and have a transition to the rollback pseudo-state.
Tag Values	None
Constraints	None
Notation	A state symbol stereotyped as «COMMIT»

Table 30: RollBack Stereotype Definition

4.5. Summary

In this chapter we presented the notation system of UTML. The notation system is not only the graphical interface of the meta-model, but it also complements its functionality by providing execution flow for transactions. It uses two types of models:

- **The Organization Model** uses the UML class diagrams and provides all the appropriate modeling elements for specifying the precise transactional semantics of the activities and it models their decomposition into other sub-activities.

- **The Execution Model** uses the UML's state machines and state-charts in order to specify the flow of execution between activities.

All modeling elements have been defined using the standard extensibility mechanisms of the UML and are completely documented.

5. IMPLEMENTATION IN ROSE AND XML TRANSFORMATION

To support web transaction design with UTML, a proper design application was built [22]. The tool has been implemented in the scope of the European IST project UWA (Ubiquitous Web Applications | IST-2000-25131), and it has been integrated with other tools that support:

- **Requirements Elicitation**. What the requirements of a Ubiquitous Web Application are and how they influence its behaviour, appearance, and customization.

- **Hypermedia Design.** How the information that a Ubiquitous Web Application communicates to the user is organized, with what semantics, and how such applications are navigated.

- **Customization Design.** How a Ubiquitous Web Application adapts to different user profiles, devices, locations and delivery channels in order to offer the same transactional functionality to the end user.

Although that this tool has not been implemented in the scope of this thesis, it provides the ability to transform UTML models into XML format and vice versa, which is part of the latter. Describing the application's functionality in XML has several advantages:

- **Documentation of Design.** Exporting UTML models into XML format is a way to document the application's design for future use by designers or other design tools. In an integrated design environment for example, transaction design information can be interchanged between tools that design the same application from a different point of view. For example, tools that design the user interface, the hypermedia structures, use cases, etc.

- **Communication of transactional semantics.** Whenever transactions of a specific web application are re-used by other systems, XML description of the precise transactional semantics can be used to facilitate their integration to these systems. Consider for example that «fligh reservation» activity of the TSS is reused by another remote application. The communication of its precise transactional semantics will make its integration and management in the new context feasible and easy.

- **Web Service Description.** Having the application's transactional functionality described in XML, it's quite easy to transform it into WSDL files. Recall that UTML describes the transactional functionality of the application and the flow of execution. This is quite similar to what web services-related languages do (WSDL, WSCL, etc.). Thus, XML description of the application's functionality enables easy derivation of web services.

- **Information Interchange between Design Tools.** Using XML format, the transaction design can be accessed by the other tools that support design for different aspects of the same application (requirements, hypermedia and customization). For example, the transactional logic of the application can be used and customized for delivering it into different terminal devices (e.g. mobile phones).

XML description of UTML models is based on an appropriate XML schema that has been defined in the scope of this thesis. This schema is presented below:

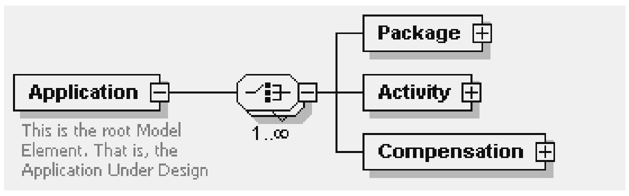

Application
This is the root Model Element. That is, the Application Under Design
1..∞
Package
Activity
Compensation

Figure 16: XML Schema: Root Level

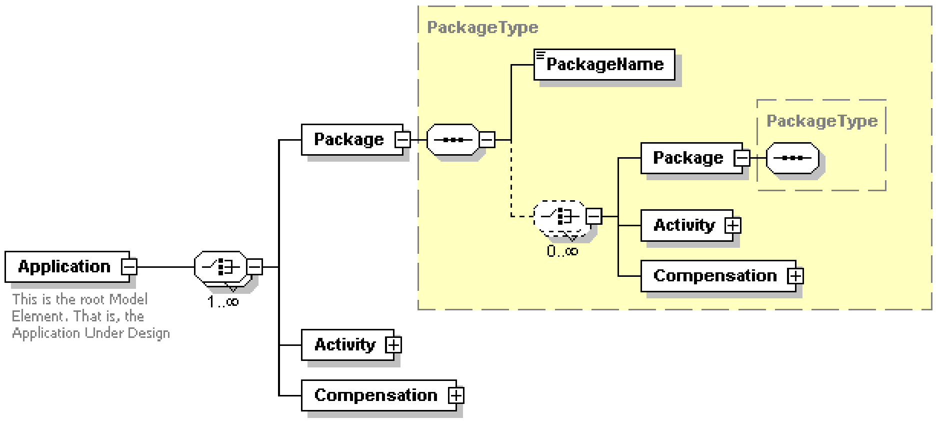


Figure 17: XML Schema: The Package Sub-Tree

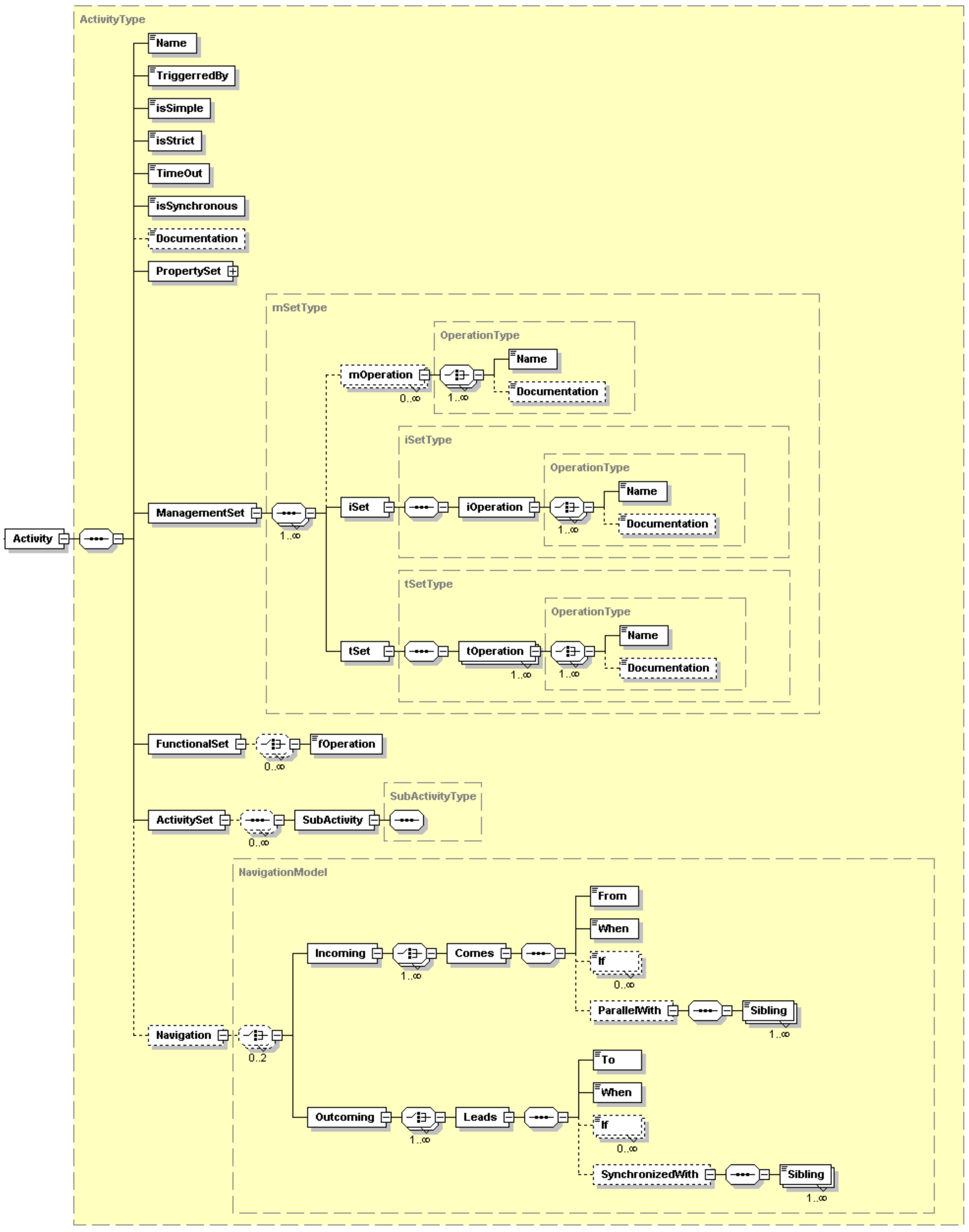


Figure 18: XML Schema: The Activity Sub-tree

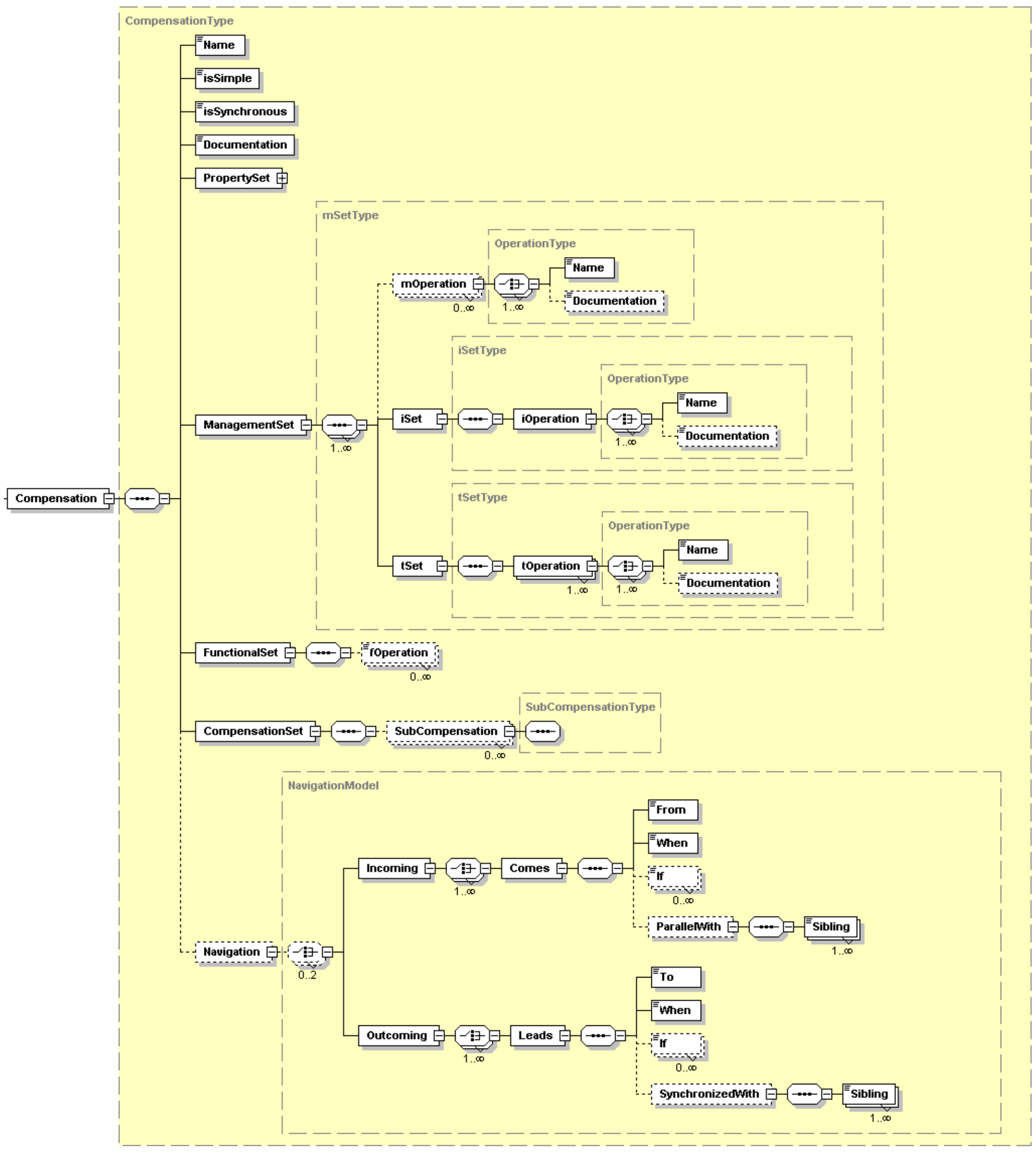


Figure 19: XML Schema: The Compensation Sub-tree

6. APPLICATIONS OF UTML

In this chapter we present examples on how UTML can be used to model complex transactions. As mentioned, UTML provides description of transactions conforming to most Extended Transaction Models, as well as modeling of custom transactions for specific application requirements.

In Section 6.1 we present the description of an Extended Transaction that conforms to the Nested Transaction Model. This model provides internal structure in a transaction but keeps this structure invisible to the outside world.

In Section 6.2 we present the description of a transaction that follows the Sagas transaction model. A Saga is set of ACID transactions that execute in a pre-defined sequence and they are semantically atomic, by executing compensating transactions in case of failure.

Finally, in section 6.3, we present a custom complex transaction that is needed for the Tourist Support System. The example, which we present in this section, accommodates different behaviors (visible and invisible sub-transactions) into the same structured transaction, incorporates «weak transactions» as part of a complex one, and it defines a complex execution flow for its activities.

CHAPTER 6. APPLICATIONS OF UTML

6.1. Describing Nested Transactions with UTML

Nested Transactions provide internal structure in a transaction but they keep this structure invisible to the outside world. A transaction in this model consists of several sub-transactions, which in turn may contain any number of sub-transactions, forming a hierarchy of transactions.

The sub-transactions of a nested transaction may commit or abort independently, subject to the following constraints. A child sub-transaction must start after its parent starts. A parent must terminate only after all its children terminate. If a parent is aborted, all its children must be aborted. However, when a child transaction fails, the parent may choose its own way of recovery.

To describe transaction of this model in UTML we have to provide the organization modeling of all transactions of the hierarchy, the well behaving rules (to define their behavior), and optionally a flow of execution which will define whether (some of) the sub-transactions will be concurrently executed or sequentially.

6.1.1. Organization Modeling of Nested Transactions

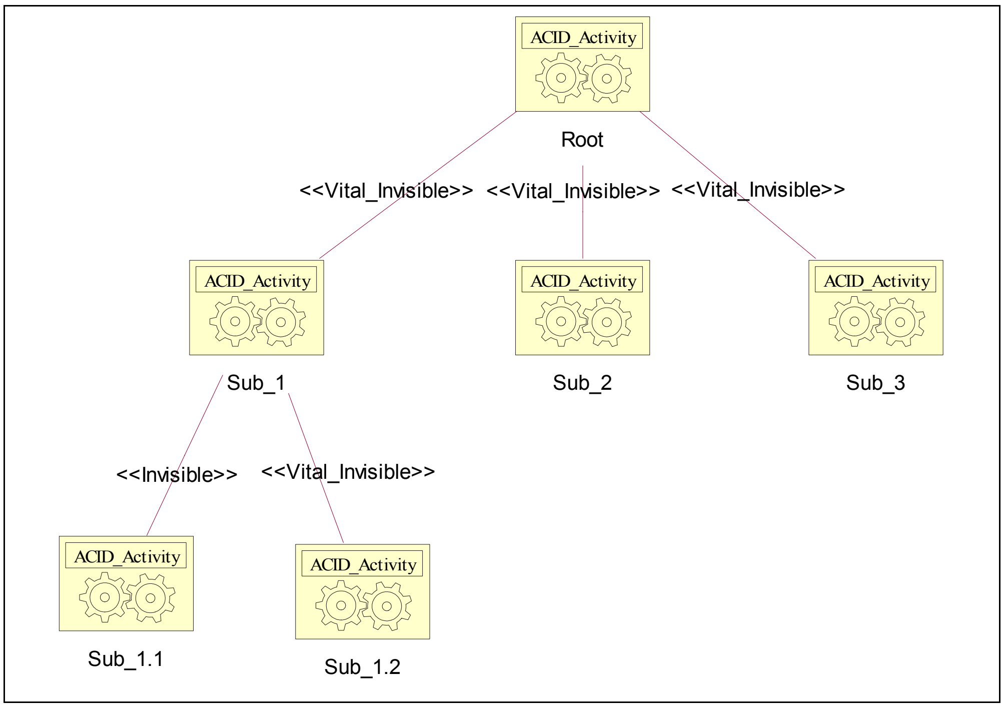


Figure 20: Structuring of Nested Transaction

The complete specification in the organization model includes the well-behaving rules that will regulate the behavior of each activity.

Activity: **Root**	
Property	**Value**
isSimple:	False
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin}
tSet	{abort, commit}
Activity: **Sub_1**	
isSimple:	False
isStrict	True
isSynchronous	True

mSet:	{}
iSet	{begin_inv_vital_sub}
tSet	{abort, delegate}
Activity: **Sub_2**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{ begin_inv_vital_sub }
tSet	{abort, delegate}
Activity: **Sub_3**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{ begin_inv_vital_sub }
tSet	{abort, delegate}
Activity: **Sub_1.1**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_inv_sub}
tSet	{abort, delegate}
Activity: **Sub_1.2**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{ begin_inv_vital_sub }
tSet	{abort, delegate}

Table 31: Activity Specifications for a Nested Transaction

6.1.2. Execution Modeling of Nested Transactions

The Execution Model in UTML provides the execution flow between activities. For the example presented above the execution flow is depicted on figure X. Note that non-vital activities may or may not be executed. UTML can describe such behavior by defining that non-vital sub-transaction are explicitly initialized by the user.

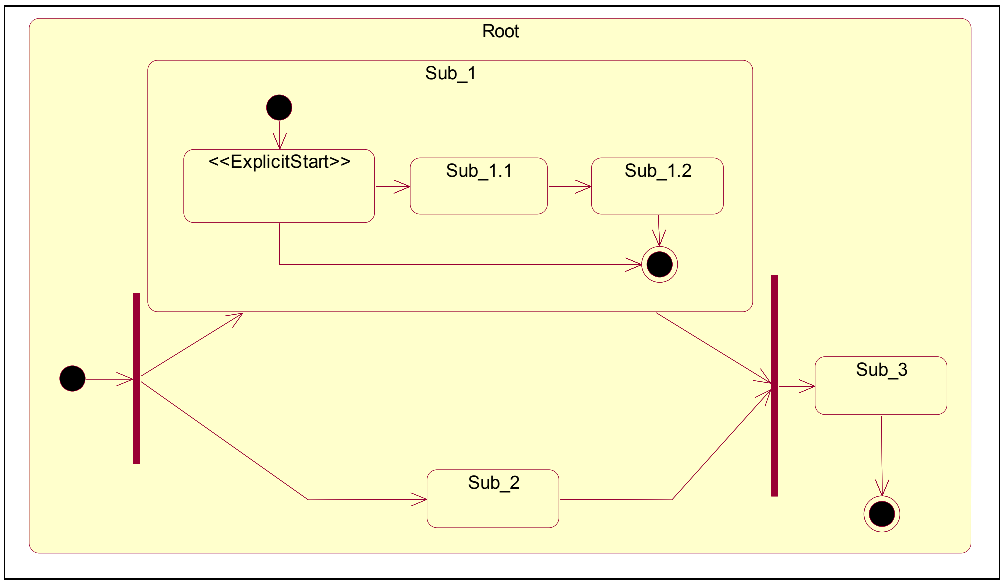


Figure 21: Execution Flow Modeling for Nested Transactions

6.2. Describing Sagas with UTML

The Sagas transaction model defines a group of ACID transactions that are executed in a predefined order. Each transaction makes its results visible to any other Saga when terminates by committing its results and releasing the resources it accessed. In case of failure, a Saga continues with the execution of the compensating transactions that must be defined for every sub-transaction. In this respect, one aspect of atomicity is achieved.

6.2.1. Organization Modeling of Sagas

The organization of a transaction conforming to the Saga transaction model is depicted on Figure 22.

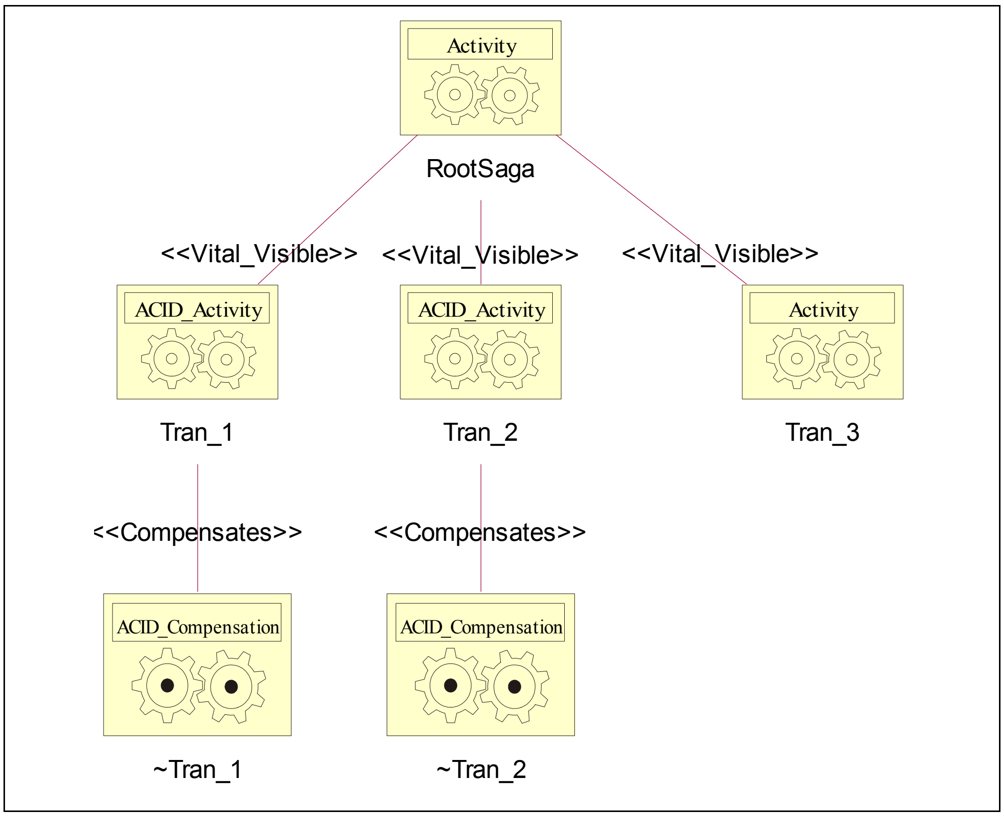


Figure 22: Structuring of a Saga Transaction

The specification of the activities involved in the above Saga transaction is shown in to table X.

Activity: **RootSaga**	
Property	**Value**
isSimple:	False
isStrict	True
isSynchronous	True

mSet:	{}
iSet	{begin}
tSet	{end}
Activity: **Tran_1**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_visible_sub}
tSet	{abort, commit}
Activity: **Tran_2**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{ begin_vital_visible_sub }
tSet	{abort, commit}
Activity: **Tran_3**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{ begin_vital_visible_sub }
tSet	{abort, commit}
Activity: **~Tran_1**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin }
tSet	{abort, commit}
Activity: **~Tran_2**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{ begin }
tSet	{abort, commit}

Table 32: Activity Specification for Sagas

6.2.2. Execution Modeling of Sagas

As mentioned, the sub-transactions of a Saga execute in a predefined sequence. The execution flow of a Saga can be modeled in UTML as shown in figure X.

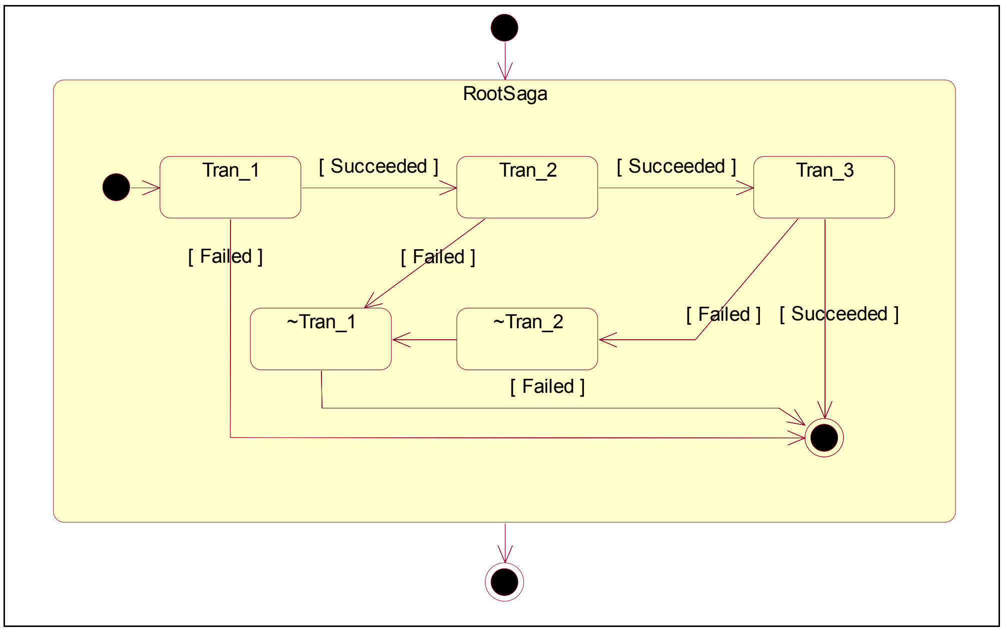


Figure 23: The Execution Model of a Saga

6.3. A Custom Transaction for the TSS

In this section we design the transactions needed in the Tourist Support System. The complexity of this example will be better understood if we discuss in detail the semantics of each task supported by this system. In this example we take the case that the user wants to plan a whole trip through the TSS.

6.3.1. User Authorization.

In order for the users to access the functionality of TSS they have to register to the system. Once registered, each time that they want to use the system's functionality they have to authorize themselves. The activity "User Authorization" is responsible to carry out this task. During this task, the users must give some information in order the system to recognize and authorize them. After that, the system has to validate this information and, if it is correct, to authorize the users.

6.3.2. Flight Reservation

The PlanTrip task includes a vital task for reserving airline tickets (vital means that this task has to be successfully executed in order for the PlanTrip to complete successfully). This task is described by the activity Flight Reservation. Because this is a quite complicated task, consisting of many discrete steps, Flight Reservation has sub-activities, each one of which is responsible for a step. To be more specific, Flight Reservation has the following sub-activities: FindFlight, SelectTicket, SupplyBillingInfo, and DebitAccount.

The activity FindFlight is responsible to find the appropriate flights. After the flight is found the system can go on with the selection of ticket. This is done by the activity SelectTicket. The first thing this activity has to do is to find out if there is an available ticket for the specific flight. If so, the activity reserves it, so as no one else can take it after the user has selected it. The final steps needed for the Flight Reservation to commit successfully have to do with the payment of the reserved ticket. During the activity SupplyBillingInfo the user is asked to to supply the system with the appropriate billing info (for example the account which will be debited). The final step is to debit the user's account with the specific amount.

6.3.3. *Hotel Reservation*

Another vital task supported by TSS is the reservation of room in a hotel. This task is described by the activity HotelReservation. As with the reservation of an airline ticket, the task of reserving a hotel room is a quite complicated one, consisting of many discrete steps. To be more specific, HotelReservation has the following sub-activities: FindHotel, SelectRoom, SupplyBillingInfo, and DebitAccount.

Comparing the names of the sub-activities mentioned above with the ones of FlightReservation we can see that are very similar. The reason this happens is because these activities perform similar tasks. Whereas FindFlight was responsible for finding appropriate flight, the activity FindHotel is responsible for finding appropriate hotels. No need to say that these acivities have the same semantics and perform nearly the same operations. The same goes on for the rest activities.

6.3.4. *Event Ticket Reservation*

A task which TSS supports is that of reserving tickets for some social events that may take place during user's vocation. This task is optional, which means the user may or may not execute it. The activity, through which this task is carried out, is the EventTicketReservation. This is another complicated activity with the following sub-activities: FindInterestingEvents, CriticServise, SelectTicket, SupplyBillingInfo and DebitAccount.

What is new in EventTicketReservation is the sud-activity CriticServise. This sub-activity allows the user to get some critic for social events by using a WebServise. It should be stressed that this service is a pay-per-use one. That is, event the user does not reserve any ticket for social events, he still has to pay for the critic he got. The rest sub-activities are similar with the ones mentioned above. In other words we have an activity which searches the wed and retrieves information about social events, another one which is responsible of booking tickets for such events and finally two more activities, one for supplying billing info and an other for debiting the user's account.

6.3.5. The Organization Model of the TSS system

The organization modeling of the entire activity structure of the TSS system, is presented into the following figures.

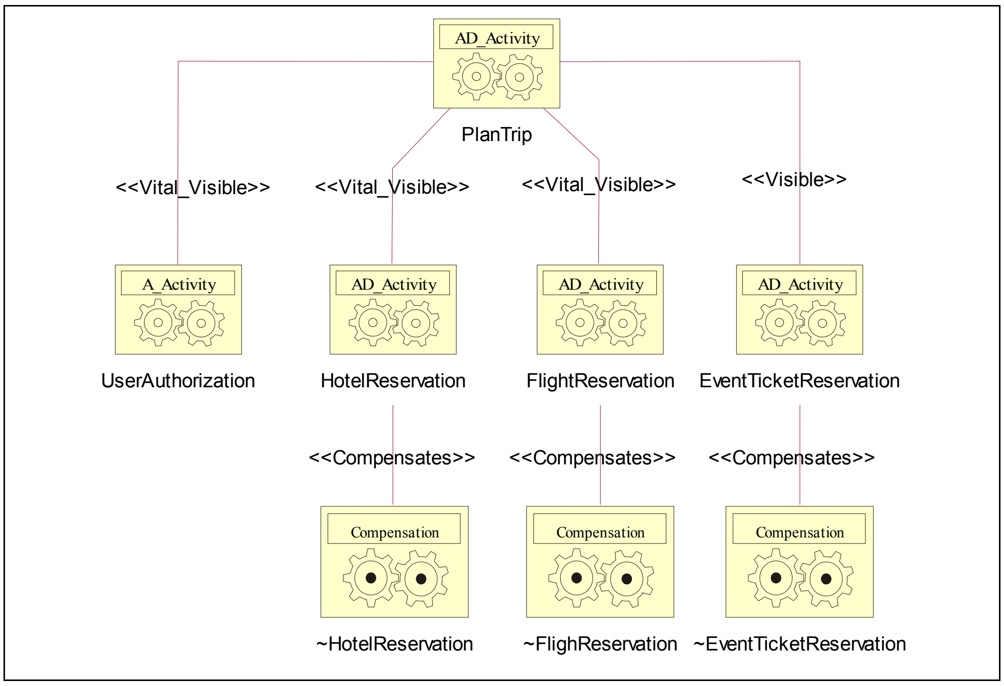


Figure 24: The Organization Model of the PlanTrip Activity

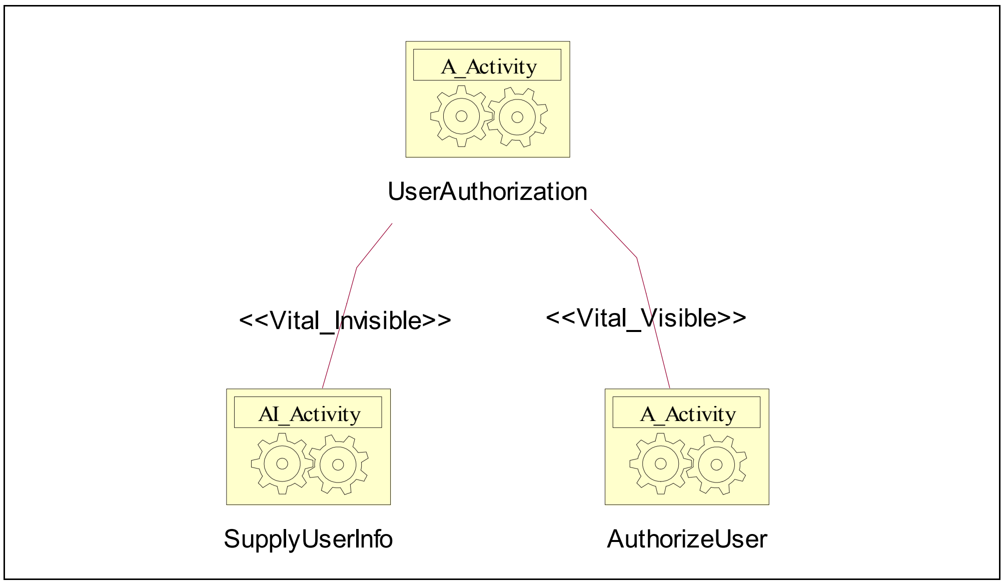


Figure 25: The Organization of The UserAuthorization Activity

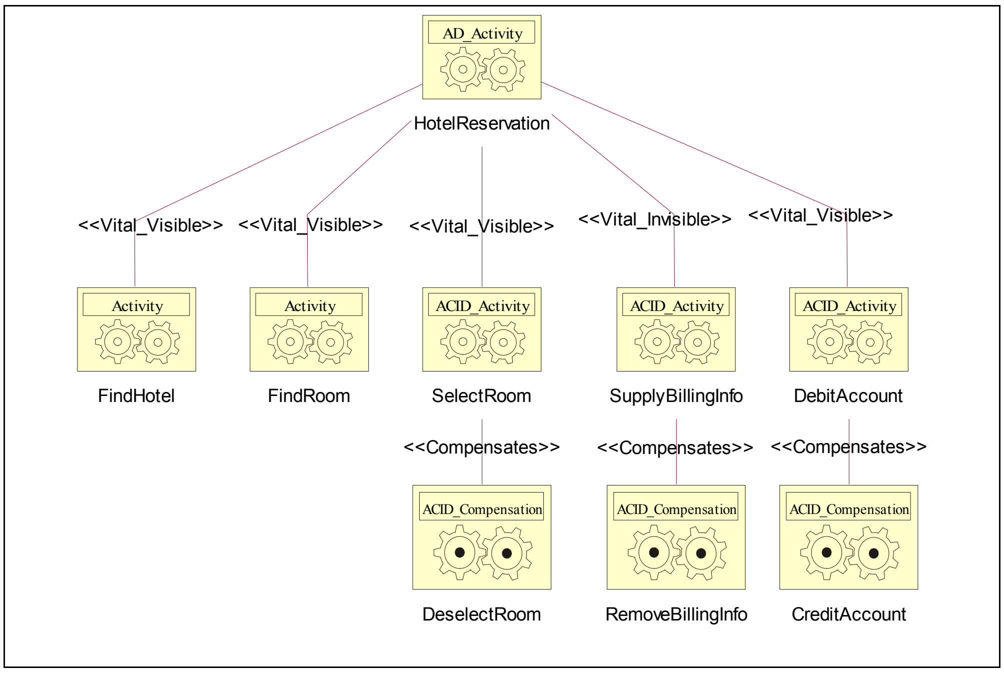


Figure 26: The Organization of the HotelReservation Activity

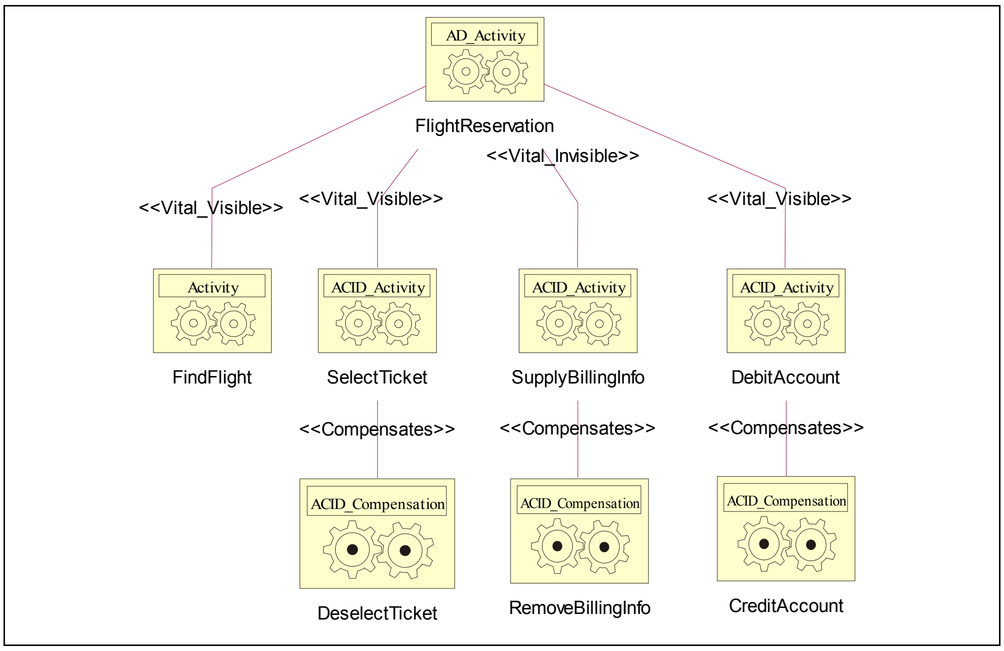


Figure 27: The Organization of the FilightReservation Activity

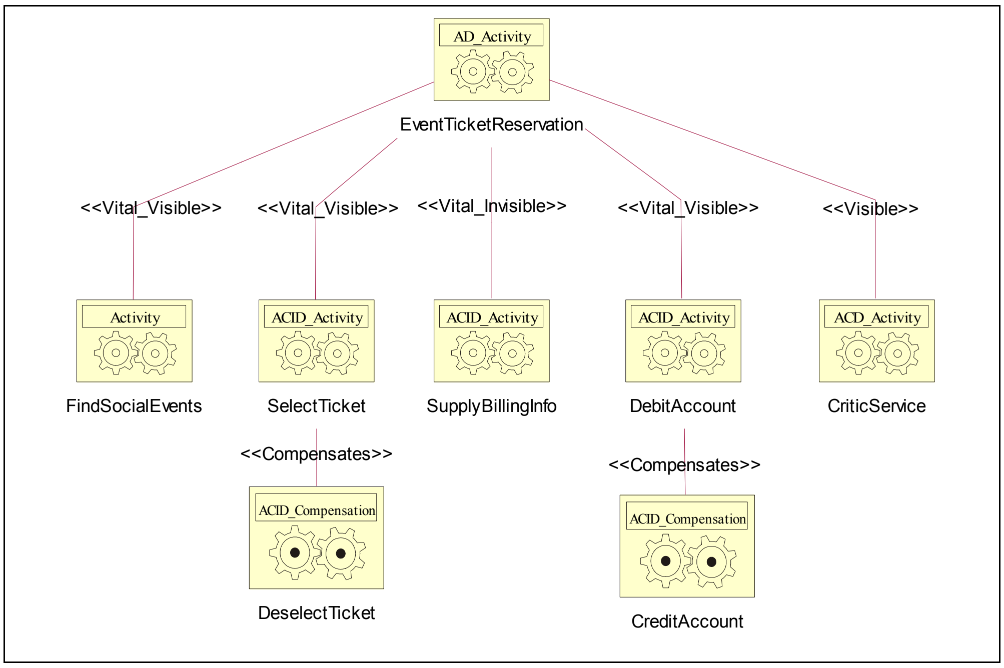


Figure 28: The Organization of the EventTicketReservation Activity

The complete specification of the activities used in the TSS is presented below:

Activity: **PlanTrip**	
Property	**Value**
isSimple:	False
isStrict	False
isSynchronous	True
mSet:	{suspend, resume}
iSet	{begin}
tSet	{end}
Activity: **UserAuthorization**	
isSimple:	False
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_visible_sub}
tSet	{abort, commit}
Activity: **HotelReservation**	
isSimple:	False
isStrict	False
isSynchronous	True
mSet:	{suspend, resume}
iSet	{ begin_vital_visible_sub }
tSet	{abort, commit}
Activity: **FlightReservation**	
isSimple:	False
isStrict	False
isSynchronous	True
mSet:	{}
iSet	{ begin_visible_sub }
tSet	{abort, commit}
Activity: **EventTicketReservation**	
isSimple:	False
isStrict	False
isSynchronous	True
mSet:	{suspend, resume}
iSet	{begin }
tSet	{abort, commit}
Compensation: **~HotelReservation**	
isSimple:	False
isStrict	True

isSynchronous	True
mSet:	{}
iSet	{begin}
tSet	{abort, commit}
Compensation: **~FlightReservation**	
isSimple:	False
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin}
tSet	{abort, commit}
Compensation: **~EventTicketReservation**	
isSimple:	False
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin}
tSet	{abort, commit}
Activity: **SupplyUserInfo**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_invisible_sub}
tSet	{abort, delegate}
Activity: **AuthorizeUser**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_visible_sub}
tSet	{abort, commit}
Activity: **FindHotel**	
isSimple:	True
isStrict	False
isSynchronous	True
mSet:	{suspend, resume}
iSet	{begin_vital_visible_sub}
tSet	{end}
Activity: **FindRoom**	
isSimple:	True

isStrict	False
isSynchronous	True
mSet:	{suspend, resume}
iSet	{ begin_vital_visible_sub }
tSet	{end}
Activity: **SelectRoom**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_visible_sub}
tSet	{commit, abort}
Activity: **SupplyBillingInfo**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_invisible_sub}
tSet	{abort, delegate}
Activity: **DebitAccount**	
isSimple:	True
isStrict	True
isSynchronous	False
mSet:	{}
iSet	{begin_vital_visible_sub}
tSet	{commit, abort}
Compensation: **DeselectRoom**	
isSimple:	True
isStrict	True
isSynchronous	False
mSet:	{}
iSet	{begin_vital_invisible_sub}
tSet	{commit, delegate}
Compensation: **RemoveBillingInfo**	
isSimple:	True
isStrict	True
isSynchronous	False
mSet:	{}
iSet	{begin_vital_invisible_sub}
tSet	{commit, delegate}
Compensation: **CreditAccount**	

isSimple:	True
isStrict	True
isSynchronous	False
mSet:	{}
iSet	{begin_vital_invisible_sub}
tSet	{commit, delegate}
Activity: **FindFlight**	
isSimple:	True
isStrict	False
isSynchronous	True
mSet:	{suspend, resume}
iSet	{begin_vital_visible_sub}
tSet	{end}
Activity: **SelectTicket**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_visible_sub}
tSet	{commit, abort}
Compensation: **DeselectTicket**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{}
iSet	{begin_vital_invisible_sub}
tSet	{abort, delegate}
Activity: **FindSocialEvents**	
isSimple:	True
isStrict	False
isSynchronous	True
mSet:	{suspend, resume}
iSet	{begin_vital_visible_sub}
tSet	{end}
Activity: **CriticService**	
isSimple:	True
isStrict	True
isSynchronous	True
mSet:	{ }
iSet	{begin_visible_sub}
tSet	{commit, abort}

Table 33: Activity Specification for the PlanTrip Transaction

6.3.6. The Execution Model of the TSS system

The execution model defined for the TSS example is described into the following figures.

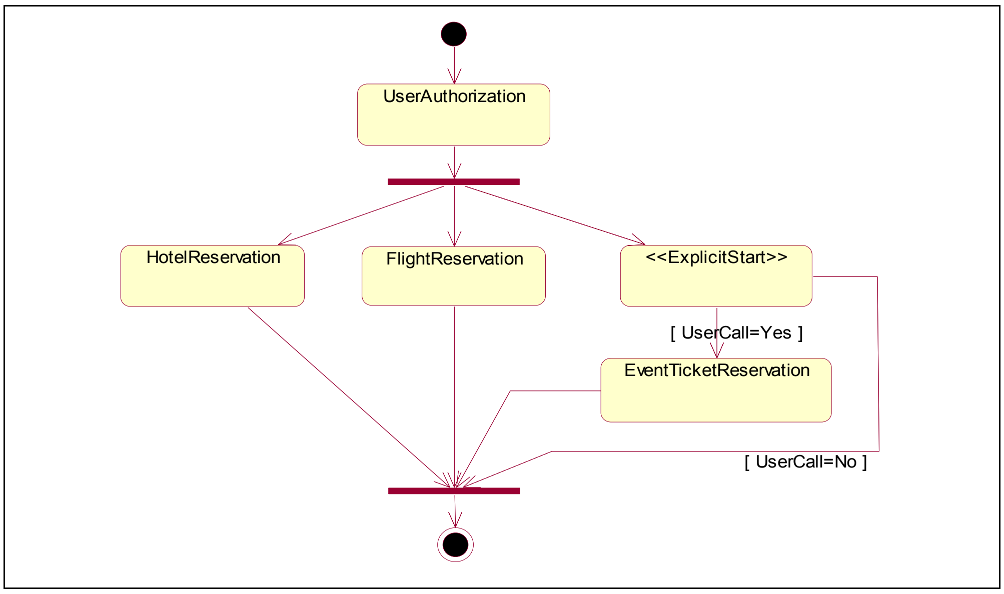


Figure 29: The Execution of the PlanTrip Activity

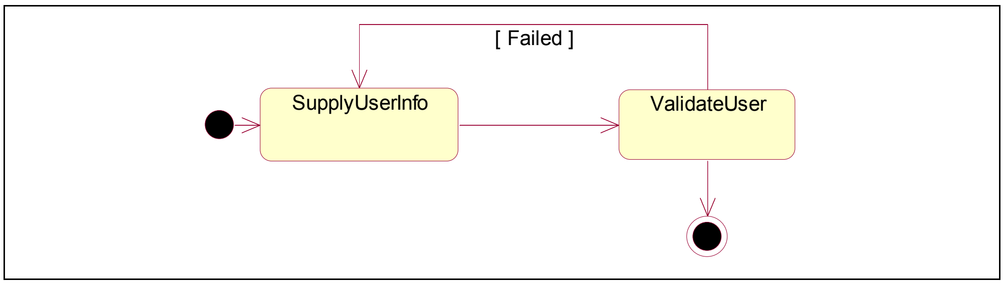


Figure 30: The Execution of the UserAuthorization Activity

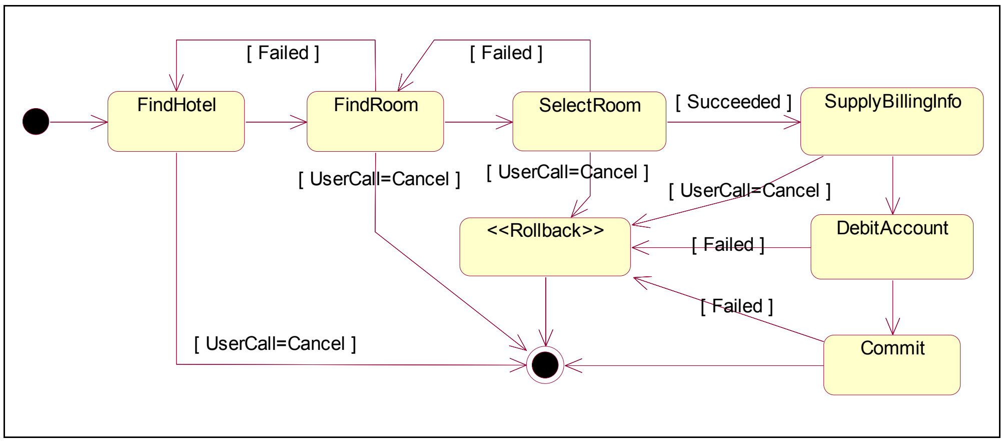


Figure 31: The Execution of the HotelReservation Activity

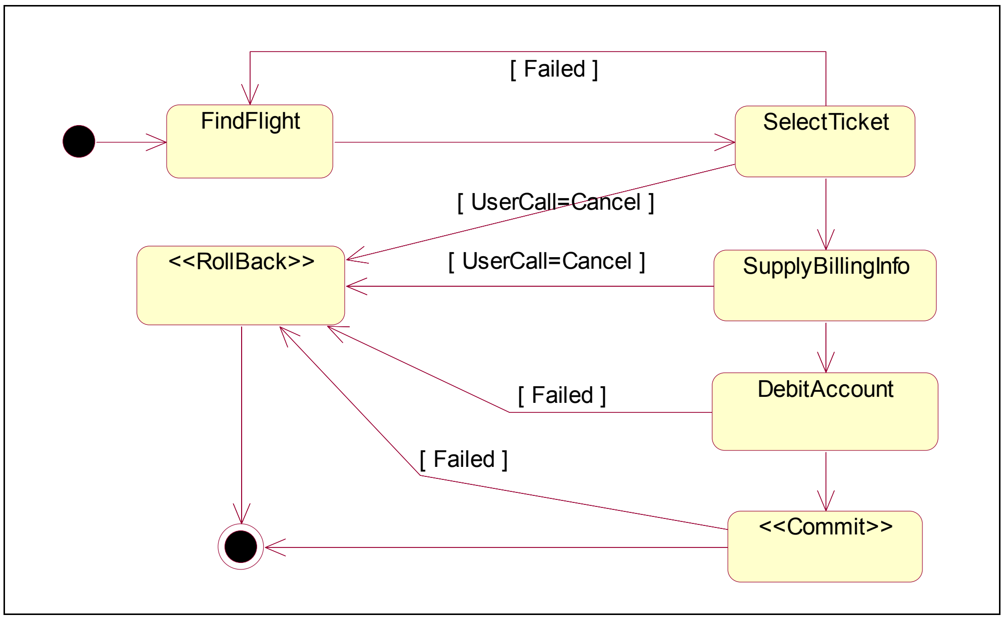


Figure 32: The Execution of the FlighReservation Activity

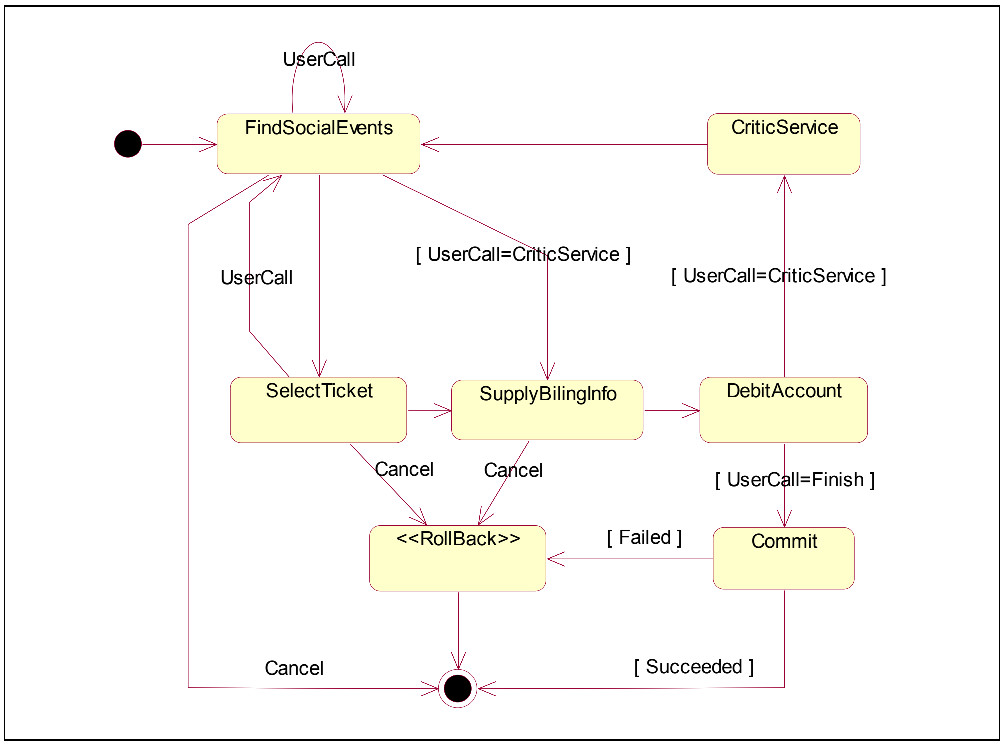


Figure 33: The Execution of the EventTicketReservation Activity

6.3.7. Summary

In this chapter we presented the flexibility of UTML in describing complex transactions. We used it to describe known transaction models, such nested transactions and Sagas, as well as to define new, custom ones appropriate for complex transactional web applications. We also demonstrated the ability of the language to define the execution of transactions and their real time dependencies.

The example of the TSS demonstrates the great flexibility of the language to accommodate different behaviors into the same structured transaction and to incorporate distributed services as part of a complex transaction. In this example we also used UTML to describe «weak transactions» that do not support the entire set of the ACID properties, which is very common to web applications.

7. CONCLUSIONS AND FUTURE WORK

7.1. Summary and Contributions

This thesis proposes UTML as a high level transaction modeling language for complex web transactions. To my knowledge, UTML is the first high level transaction modeling language in the literature and is completely compatible to Unified Modeling Language (UML) standard of the industry.

Web transactions may be hierarchically structured, consisting of several sub-transactions, each one accessing different distributed and diverse resources or utilizing pre-existing logic or services. The divergence of resource interfaces and pre-existing functionality's semantics impose several requirements that the used transaction model should meet.

Although several extended transaction models have been proposed in the literature, no one can provide the great flexibility that web transactions require. Their limitation comes mainly from their inflexibility to accommodate different behavioral patterns in the same structured transaction. Each transaction model defines a specific behavior that all transactions of a complex hierarchy have to follow. This «monotony» of complex transaction structures makes difficult the integration of diverse resources into the same structure, or the access of resources that do not satisfy the requirements that this behavior sets. Thus, UTML has opted for the use of meta-model for modeling transactions for web applications.

UTML consists of two main parts. A ***transaction meta-model*** that provides the basic modeling concepts, their relationship and their specification, and a ***notation system*** that makes its meta-model easily applicable and handy in the design process of complex web applications. Moreover, the notation system complements the transaction meta-model by providing execution flow modeling of transactions, defining at the same time a primitive user navigation model.

The main advantages of UTML can be summarized into the following:

- ***High level transaction modeling***. It makes the complex modeling concepts of a rich transaction meta-model applicable through a high level notation system that it is compatible to UML.

- ***Modeling of both the static structure and dynamic behavior of transactions.*** In web applications, it is important to model the flow of execution that has to be respected by both the user and the system in order to avoid the user confusion and violation of the transaction properties.

- ***Description of «weak transactions».*** In web applications, not all activities need to support the entire set of the ACID properties. Also, most of the legacy databases and the file systems that are used as back storage (resources) do not support them. UTML provides the ability to arbitrary define which of these properties will be supported by an activity and moreover, it provides mechanisms to check the correctness of this assignment.

- Accommodation of diverse semantics and behaviors into the same transaction model. This is provided by explicitly defining the management of each activity and the decomposition semantics between a parent activity and each of its children.

- Description of new transaction models, by using its extensibility mechanism. New transaction models can be defined by extending the management operations that the language provides and defining appropriate well-formedness and well-behaving rules that precisely describe the newly defined model's behavior.

- Transaction design in both top-down and bottom-up fashion. In a top-down design fashion the designer successively decomposes complex activities into smaller and simpler ones, while in a bottom-up fashion it takes into account limitations and divergence of resources to synthesize complex activities that properly accommodate the different semantics and interfaces of diverse resources or pre-existing functionality.

- Description of transactions conforming to the most of the well-known transaction models. The utilization of transaction models that have already been proposed in the literature is many times needed in web applications and UTML provides modeling of transactions that conform to the most of these transactions

- Documentation of the precise transactional semantics that a web application has. UTML can be used to document web applications which exhibit complex transactional behavior. Documentation of the precise semantics of the application's logic enables easy derivation of new application view that will be used to deliver the application's functionality through different devices, channels, etc.

- Description of transaction design in XML format. The description of application's logic and transactional behavior into XML format makes possible the communication of the application's semantics into other co-operating applications. Also, having the application's transactional logic described in XML format, the export of the application into the outside world as one or more web services becomes possible and easy (using for example an XSLT to transform XML to WSDL).

7.2. Future Work

In this section ideas for future extensions of UTML are presented along with a brief description of each one. It should be noted that each one of these extensions could provide the basis for new research activities.

- **Modeling of Dataflow Dependencies** between transactions that belong to the same structured transaction. Defining these dependencies between transactions may lead to additional functionality of UTML concerning:
    - **Flexible Compensation Strategies.** The definition of flexible compensation strategies for transactions may prevent the compensation of an entire composite transaction that is to be re-executed. The abortion and re-execution of a transaction does not imply that all executed sub-transactions have to be abrogated. The modeling of dataflow dependencies between sub-transactions of a complex structure may prevent unnecessary loss of work, by identifying which of the successfully executed sub-transactions have to be compensated.
    - **Advanced Concurrency Algorithms.** By identifying the dataflow dependencies between transactions the cooperative ones can be identified and this identification may lead to advanced concurrency algorithms for specific transaction models.
- **Description of Asynchronous Transaction Execution.** Web transactions, and especially those that are to be executed on mobile devices, may be executed asynchronously. The asynchronous execution

may imply data replication, allotment or even virtual execution of some transactions at disconnected mode. When those transactions are to be reflected back to the central web application, a synchronization process is usually needed. An appropriate extension of UTML could precisely describe such transactions in detail and provide new models for managing their execution and behaviour could be defined.

- **Modeling of Persistent Transactions.** The execution of long-lived transactions is typically unsheltered to failures. Also, in mobile execution environments the frequent failures and the poor user typing capabilities make the re-execution of transactions a redundant task that should be avoided. To avoid this situation, persistent transactions should be properly defined with the sense that the transaction itself is recoverable (after a failure, it recovers to the state it had before this failure). Of course this is not always feasible. However, it could be of high interest to thoroughly investigate this possibility.

BIBLIOGRAPHY

[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", Kluwer Academic Publishers, 1998

[3] A. Cockburn: "Writing Effective Use Cases", ISBN: 0201702258, Addison-Wesley, 2000

[4] BPMI: "Business Process Modeling Language", www.bpmi, 2002

[5] 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

[6] 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

[7] F. Casati, M. Sayal, M. Shan: " Developing E-services for Composing E-Services", In the Proceeding of the 13th International Conference on Advanced Information Systems Engineering, CAiSE, 2001

[8] 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

[9] G. Booch: "The Unified Modeling Language User Guide", ISBN: 0201571684, Addison Wesley, 1998

[10] G. Weikum, H.-J. Schek: "Concepts and Applications of Multilevel Transactions and Open Nested Transactions", in Database Transaction

Models for Advanced Applications by A. K. Elmagarmid, Morgan-Kaufmann, 1992.

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

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

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

[14] [H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the 16th International Conference of VLDB*, 1990].

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

[16] IBM: "Web Services Flow Language - WSFL 1.0", www.ibm.com

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

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

[19] K. Schwarz, C. Turker, G. Saake: "Transitive Dependencies in Transaction Closures", in the Proceedings of the International Database Engineering and Applications Symposium, 1998.

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

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

[22] M. Kozyri: “Implementation of a Tool to Support Design of Web Transactions with UTML”, Technical University of Crete, 2002

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

[24] Object Management Group, “Activity Service Specification”, www.omg.org, 2001

[25] Object Management Group, “Object Transaction Service Specification”, www.omg.org, 2001

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

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

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

[29] 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

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

[31] P. Lewis, A. Bernstein and M. Kifer: “Databases and Transaction Processing. An application –Oriented Approach” ISBN 0201708728, Addison Wesley, 2002

[32] Q. Chen, U. Dayal, M. Hsu: “Conceptual Modeling for Collaborative E-business Processes. In the Proceedings of the 20th International Conference on Conceptual Modeling, 2001

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

[34] S. Si Albir: "UML In A Nutshell", O'Reilly & Associates, ISBN: 1565924487, 1998

[35] U. Dayal, M. Hsu, R. Ladin: "Business Process Coordination: State of the Art, Trends, and Open Issues, In the Proceedings of the 27th conference on VLDB, 2001

[36] W3C: "Web Services Conversation Language", W3C Note, www.w3c.org, March 2002

[37] W3C: "Web Services Description Language", W3C working draft 9. www.w3c.org, July 2002