Models and Meta Models for Transactions in Web Applications

Mark Douglas Jacyntho[1,2,3] and Daniel Schwabe[1]

[1] Departamento de Informática, PUC-Rio, Av. Marquês de São Vicente, 225, Rio de Janeiro, Brazil
{mjacyntho,dschwabe}@inf.puc-rio.br
[2] Universidade Candido Mendes, Núcleo de Pesquisa e Desenvolvimento, Rua Anita Pessanha 100, Campos dos Goytacazes, RJ, Brazil
[3] Instituto Federal de Educação, Ciência e Tecnologia Fluminense, Av. Dário Vieira Borges, 235, Bom Jesus do Itabapoana, RJ, Brazil

Abstract. In this paper, we present a DSL to specify business and Web transactions in a systematic way, addressing both informational and behavioral perspectives. Our meta-model is based on the reification of transactions, where transactions are modeled as first-class types, supporting attributes, associations, operations and state machines. Treating transactions as domain type instances facilitates the interplay with other models, such as the navigational model, which is a view of the domain type model.

Keywords: Business Process, Workflow, Business Transaction, Workflow Transaction, Web Transaction, Conceptual Modeling, Process Modeling, Transaction Modeling, Meta-model, Ontology, Semantic Web, DSL, MDE, MDWE, MDD, MDA.

1 Introduction

Web applications are multi-concern systems. Since each concern has its own characteristics and peculiarities, a good modeling strategy is to promote their clear separation. This approach is adopted in the majority of Web development methodologies. In general, for each concern, there is a separate model: domain model, navigation model, interface model, behavior model, and so on.

Web application behaviors have evolved from simple request handling mechanism to highly complex behavior, within which data and transactions must be managed to support intra- and inter-organization business transactions.

Current Web applications are developed to carry out tasks or goals that are part of a well-defined business transaction workflow, in accordance to its rules and constraints, serving different users whose joint work needs to be coordinated. Issues like distribution and concurrency must be considered, and conflicts solved. For example, in an e-commerce Web site two concurrent sessions can be simultaneously checking out an order containing the last unit of the same product, and last session to send the confirmation will not have success in acquiring the product (that was available only a fraction of

F. Daniel and F.M. Facca (Eds.): ICWE 2010 Workshops, LNCS 6385, pp. 37–48, 2010.

time earlier). Another typical situation could be the concurrent reservation of the same seat in a flight at an airline Web site.

Following [7], within a business transaction one or more systems are used and for each system one or more system transactions are triggered. In the specific case of Web applications, these system transactions can properly be called Web transactions. Based on [3] a Web transaction can be defined as "a sequence of activities performed by a user to achieve a specific goal related to a step of a business transaction, through a Web application".

Furthermore, another very interesting issue in Web applications is how to combine Web transactions and navigation. Strictly speaking, navigation is a behavior just like any other application behavior. The only difference is that its semantics are known ahead of time, as provided by the standard hypertext node-and-link navigation models, instantiated by the Web. Because of this, specialized "navigation models" have been proposed, to simplify the specification of this portion of the application's behavior. Typically, navigation is implemented as a read-committed nested transaction to support the selection of items to be processed by a subsequent sibling transaction. For example, the Web transaction "Place Order" has two sequential nested transactions: "Items Selection" and "Checkout". The former consists in navigation over the catalog to select items to buy and the later creates an order with the previous selected items. Like any other transaction, navigation is subject to concurrence interferences - the information can become obsolete. This fusion of paradigms is a complex task that needs to be carefully analyzed during the process of Web application design, at the risk of unwanted behavior and results.

As the transactional concern is clearly present, and thus a specific model for Web transactions is needed. The lack of this model makes transactions to be relegated to a secondary level in the implementation code – e.g., in the controller´s code in MVC architectures. Each designer creates his ad-hoc controllers based only on the notion of user session and requests along this session. In fact, along the user session there are several Web transactions taking place, with well defined properties, parameters and execution flow, nevertheless these issues are not explicitly modeled. Which request starts the transaction? Which commits? Inside the scope of which transaction a request occurs? In addition, the necessary information (objects) to carry out the transaction are commonly stored, indiscriminately, inside the user session, using a dictionary style cache mechanism, where objects are stored using an associated key. Since these objects are the transaction parameters, why not reify the transaction concept and model these parameters as transaction properties? The transaction object would be itself the information cache and this object could be treated as any other domain object (persistable, navigable, updateable, and so forth).

In this paper, we propose a DSL (defined by a meta-model and a notation) to address the explicit modeling of business and Web transactions, reifying the concept of transaction in a first-class meta-model type. Transaction-type instances correspond to transaction occurrences and store all the state of that occurrence. At runtime, transaction instances are activated via one controller, and implement the transaction logic using its instance data. It is important to say that our focus is Web transactions, not information (data) tier transactions (e.g. Databases). We assume the existence of a transactional information tier.

The challenge of this work is to offer a meta-model that is complete, but without restricting the transactional behaviors that the designer may want to model. It should be possible to model from traditional ACID transactions to modern non-ACID transactions (for example, transaction which permits lost updates that could be solved semantically by another model).

In the same manner we have been treating the navigation concern, from now on, with the transaction primitives defined by the meta-model, we can explicitly model the transactional issues that were formerly addressed only as an implementation detail.

This paper is structured as follows: in section 2 we introduce the meta-model of the DSL. Next, in section 3, we present an example and some observations regarding the benefits of using the DSL. Section 4 presents some related work and, finally, section 5 concludes this paper with final remarks and future work.

2 DSL Meta-model

Our meta-model is based on the notion of reification, by promoting the concept of transaction to a first-class type, responsible to keep the execution state (actual parameters, local variables, execution status, etc), controlling all the execution steps. Reified transaction types can be used like any other domain type, and may also be persisted.

Another significant characteristic of this meta-model is the clear separation, into distinct meta-classes, of the transaction specification ("what") and its possible enactments ("how"). Specification here means a contract given the pre/post-conditions added through domain invariants. Enactment means one or more execution protocols that satisfy the contract, achieving the same post-condition. Each enactment is described by a flow model, where the transaction is further detailed into increasingly finer transactions until the lowest level of abstraction is reached, i.e. operation calls. Note that to reuse one transaction as a nested one inside an enactment´s flow of a parent transaction, we need only to know the specification of the invoked transaction ("what"), and any enactment ("how"), instantiated at runtime, will work fine, because, by definition, it satisfies the contract specification.

The meta-model has two orthogonal perspectives. The first is the informational or static view regarding the properties (attributes and associations) of the transaction and the second is the behavioral or flow view, where the focus is on the execution flow of a possible enactment that fulfills the transaction contract.

Along the explanation, the concepts will be clarified using excerpts of a case study about a business transaction dealing with analysis of artifacts (*papers*, *workshops*, and *talks*) submitted to a conference edition, encompassing: submission, review and selection of artifacts.

2.1 Informational Perspective

The informational perspective is concerned with the necessary information - parameters and local variables used to carry out the transaction - and objects manipulated

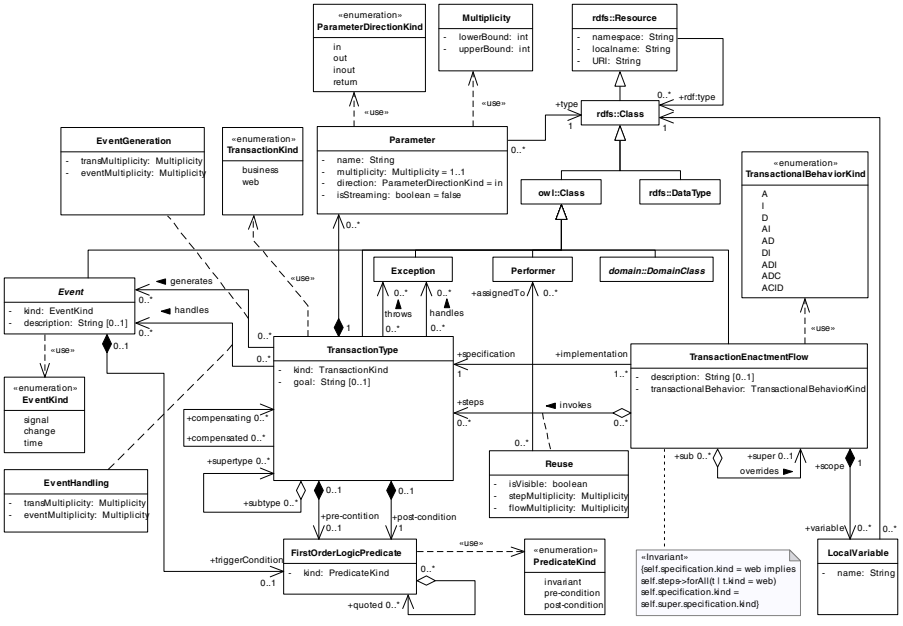

Fig. 1. Meta-model informational perspective

during the course of the transaction. Figure 1 shows the informational portion of the meta-model as a UML-style class diagram.

The core concept of this meta-model is *TransactionType*, which models a transaction specification ("what"). There are two kinds of transactions: *business* and *Web*.

To specify a transaction, we must define its contract, encompassing *Parameters*, *Exceptions*, and *pre/post-conditions*. *Parameter* is used to model the input/output parameters of the transaction. Following the design by contract approach, *post-condition* is a first order logic predicate (Boolean expression) that specifies the effect of the transaction that must be guaranteed by any enactment.

Sometimes the post-condition makes sense only under certain initial conditions, so called *pre-conditions* which are also Boolean predicates. If the pre-condition is not true when a transaction starts, this specification does not apply to it, so the outcome is unspecified, and the post-condition is not guaranteed. Invariants may also be defined that are automatically added (logic operator AND) to the pre/post-conditions of all defined transactions.

Finally, the exceptional outcomes of a transaction can be specified using *Exceptions*. An exception can be considered a special kind of return parameter. For each exception there may be a transaction that plays the role of exception handler.

To illustrate these primitives, the Web transaction used to submit the artifact to a conference edition - "Artifact Submission Transaction" - has three input parameters (*authors*, *first author*, *conference edition*), one output parameter (*new artifact*), one exception (*artifact wrong format exception*). The pre-condition is that the current date

has to be before the edition´s deadline submission date and the post-condition is that a new artifact instance has been created, with status *submitted*.

Another important primitive is the notion of *Event*, which models events that the application sends or receives addressed to or originated from external entities (actors). An event can be compared to a trigger that initiates a transaction. For instance, the author decides to withdraw his artifact. An incoming "Withdrawal Event" triggers the Web transaction "Artifact Withdrawal Transaction", suspending any other transaction that may be in progress ("Reviewers Assignment", "Obtaining Artifact To Review", "Artifact Review Submission").

The transaction type may have subtypes. All specifications of the supertype transaction are inherited by the subtype transactions. This is a type or specification inheritance and, thus, overriding is not permitted, only extensions. Based on the design by contract rules, a subtype post-condition can only extend the supertype post-condition, not redefine it. In addition, the subtype pre-condition must not extend the supertype pre-condition. The subtype pre-condition can be weaker but not stronger than the supertype pre-condition. In other words, like any other kind of subtype, a transaction subtype must satisfy the *Liskov Substitution Principle* [6], so that a subtype instance can properly be used in any context where we reason just using the supertype specification, without knowing anything about possible subtypes. In our case study, for each subtype of *Artifact* (*Paper*, *Workshop* and *Talk*), there is a corresponding Web transaction for submission which is a subtype of the "Artifact Submission Transaction".

Finally, a transaction may have one or more alternative compensating transactions. A compensating transaction is a way to “reverse” the effects of a terminated transaction in case of a subsequent abort. It is different from rollback because the effect is not rolled back, but replaced by another effect that compensates the first.

Once the transaction is specified, the next step is to model at least one enactment (*TransactionEnactmentFlow*) of "how" to achieve its post-condition, that is, one possible realization or implementation of the transaction ("what"). Statically speaking, an enactment defines the ACID properties and its constituent nested transactions (invoked/reused) that define the steps of the referred transaction. It is perfectly possible to define ACID and non-ACID enactments. Each enactment defines what nested transactions are (re)used to carry out the specified parent transaction and what *Performers* (actors) can execute them. For example, as mentioned before the business transaction of artifact analysis - "Artifact Analysis Transaction" - encompasses submission, review and selection of artifacts. Therefore, one possible enactment of this business transaction - "Artifact Analysis Enactment" - has the following nested Web transactions as steps: "Artifact Submission", "Reviewers Assignment", "Obtaining Artifact to Review", "Artifact Review Submission", "Artifact Decision", "Artifact Final Version Submission", "Artifact Withdrawal", and "Wrong Format Exception Handling". These last two ones are, respectively, the event handler associated with the "Withdrawal Event", if the author decides to withdraw his artifact, and the exception handler associated with the "Artifact Wrong Format Exception", exception thrown by the "Artifact Submission" Web transaction, if the submitted artifact does not adhere the conference required format.

An important point is the constraint that Web transactions can only be composed of other Web transactions. It does not make sense to put a business transaction inside a Web system transaction. Another important point is to indicate whether the outcome of the terminated nested transaction is visible or not beyond the parent transaction boundary.

Finally, an enactment can be defined via inheritance of another enactment. This is an "implementation" inheritance and, therefore, extensions and overriding, both are permitted. Nevertheless, there is a restriction that an enactment can only inherit from another enactment of the same transaction kind (business or Web). Again, in our example, for each subtype of *Artifact* there is a corresponding enactment which inherits from "Artifact Analysis Enactment", because of the extra constraints, like for example, the corresponding Web transaction "Artifact Submission" subtype: for *Paper* must be "Paper Submission", for Workshop must be "Workshop Submission" and for *Talk* must be "Talk Submission".

To specify the dynamic perspective, *TransactionEnactmentFlow* allows modeling the flow logic of the transaction´s steps, describing the sequence in which the nested transactions take place. This is the subject of the next section, where we explain the behavioral perspective of the meta-model.

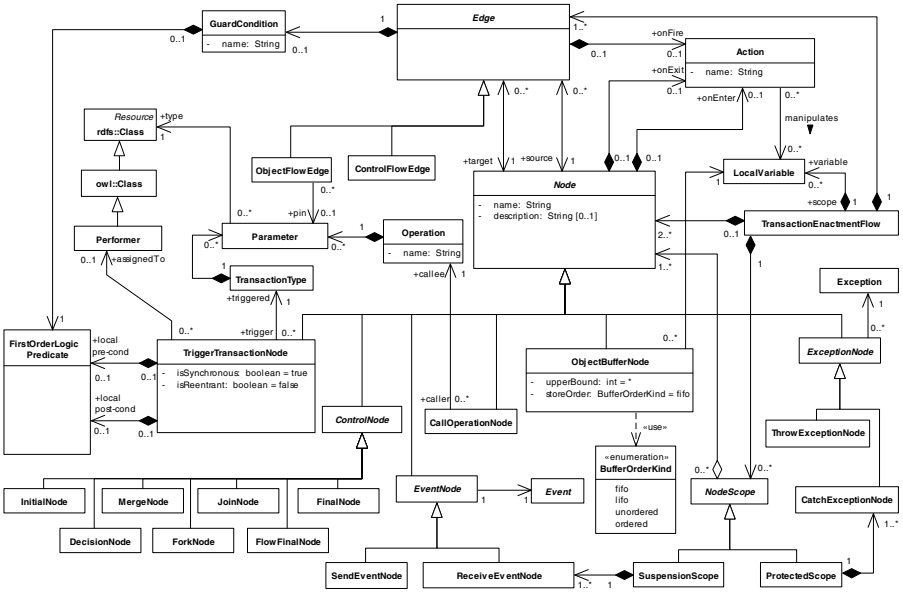

Fig. 2. Meta-model behavioral perspective

2.2 Behavioral Perspective

This view defines the possible execution flows between steps (nested transactions or operations) of a transaction enactment. This part of the meta-model is a specialized version of the UML 2.0 activity meta-model, which is based on Petri nets [4].

Basically, the *TransactionEnactmentFlow* primitive contains nodes connected by edges forming a complete flow graph, and control and object (data) tokens flow along the edges and are operated on by nodes, routed to other nodes, or stored temporarily. Control token is a Boolean data that indicates that that target activity node cannot start until the source activity node terminates. Figure 2 shows the behavioral part of the meta-model. There are meta-concepts in common between the informational and behavioral diagrams of the meta-model. To simplify the diagram, some relationships already shown in Figure 1 were omitted in Figure 2.

The informational part of the *TransactionEnactmentFlow* models only the static composition of the enactment; its constituent transactions might happen in parallel, in sequence, there might be alternative paths, repetitions, etc. The role of the behavioral meta-model is to complement the definition of a transaction enactment, establishing the execution flow.

There are six kinds of node: *TriggerTransactionNode* - represents a transaction trigger. There may also be a *Performer* associated who is the actor (role) that executes this transaction; *CallOperationNode* - represents a call to an operation; *Control Nodes* - route control and objects (data) through the graph; *ObjectBufferNode* - holds data values (objects). Represents the usage of a *LocalVariable* as a buffer; *Exception Nodes* - throw or catch an exception; *Event Nodes* - send or accept an event.

In addition to node and edges, there is the notion of *Scope*, which is a set of nodes grouped for some purpose. There are two kinds of *Scope*. *SuspensionScope*: this scope gathers *ReceiveEventNodes* so that when the event happens any activity inside the scope is suspended. *ProtectedScope*: this scope is similar to the try/catch mechanism in programming languages. The nodes inside the scope are protected against exceptions thrown within the scope. The scope has one or more *CatchExceptionNodes* associated and the scope is protected from each corresponding *Exception*.

Given this meta-model, it should be clear that it is possible to define an enactment which implements a standard hypermedia navigation engine, which keeps track of the navigation state of the application at any point. From this, it straightforward to see that navigation becomes just another transaction in the application.

3 An Example

As we know, given an application domain, there exist common features present in the vast majority of the models that may be used to build a standard reusable framework model. Therefore, in general, the modelers do not instantiate a meta-model from scratch, but create their models by extending pre-existing standard models (abstract meta-model instances).

To assist the meta-model instantiation, we provide a standard base instance of our meta-model to be extended by a particular model. This standard abstract model, not shown for the sake of space, defines the *Transaction* abstract class from which all transaction classes can be derived. In addition there is the class *Flow* (*Thread*) that represents the execution flows of the transaction. One transaction can have one or more executions flows or threads. Two or more flows are created when there is a fork control node in the behavioral model. Each *Flow* maintains the steps (Transactions

instances) executed in its scope, including its current step. One Transaction can have many flows in execution, but only one flow is effectively running (progressing), per user session.

The classes have some standard properties and an execution status attribute. The Flow class has five lifecycle statuses: *ready*, *running*, *blocked*, *finished*, and *interrupted*. The Transaction class has duration attribute and the following statuses: *definingActualParameters*, *executing*, *committed*, *aborting*, *aborted*, *compensating*, *compensated*, and *terminatedByException*.

We next show our case study for a business transaction of analysis of artifacts submitted to a conference as an extension of the standard model. In this example, this business transaction is called *ArtifactAnalysis* and one possible enactment has seven Web transactions: *ArtifactSubmission* - submission of the artifact executed by one of the authors; *ReviewersAssignment* - assignment of reviewers to assess the artifact, carried out by the pc-chair; *AcquiringArtifactToReview* - each reviewer obtains the artifact to evaluate; *ArtifactReviewSubmission* - each reviewer submits the respective evaluation; *ArtifactDecision* - the pc-chair accepts or rejects the artifact; *ArtifactFinalVersionSubmission* - if the artifact was accepted, the author submits the final version; *ArtifactWithdrawal* - the author withdrawals the artifact.

For the sake of space, the domain model will not be shown. The main domain concept is *Artifact* and its subtypes (*Paper*, *Workshop*, and *Talk*). Among other properties, *Artifact* has *Authors* and status (*submitted*, *waitingReviewers*, *inReview*, *reviewed*, *acceptedWaitingLastVersion*, *withoutLastVersion*, *accepted*, *rejected*, *and withdrawn*). Each *Artifact* is associated (submitted) with one conference *Edition*. Each *Artifact* undergoes three *Reviews*, each made by one *Reviewer*. Finally, the Artifact is accepted or rejected by the *PC-Chair* of the *Edition*. If accepted, a final version must be submitted, otherwise the *Artifact* is not considered accepted. The first step is to model the informational view of the transactions. Figure 3[1] shows informational view of the business transaction *ArtifactAnalysis* and of the Web transaction *ArtifactSubmission* (suppressing its pre/post-conditions, for lack of space) and its subtypes. To simplify, we show only one possible enactment (*ArtifactAnalysisEnactment*) of the *ArtifactAnalysis* specification and we do not show the ACID enactments of the *ArtifactSubmission* Web transaction and its subtypes. The ACID behavior of the *ArtifactAnalysisEnactment* has only "D" (Durability) and, remember that this enactment must honor the post-conditions "note" associated with the *ArtifactAnalysis* transaction. The dashed line subtype symbol between *ArtifactAnalysisEnactment and ArtifactAnalysis* denotes realization (one possible implementation) of the transaction. This is the type classification of the enactment, not an inheritance, that is, all instances of one enactment are instances of the corresponding transaction. Note that *ArtifactAnalysis* and *ArtifactSubmission* are subclasses of the *Transaction* class defined in the standard abstract model.

---

[1] DSL notation: double oval - *Business Transaction*; simple oval - *Web Transaction*; rounded rectangle - *Enactment*; up/down arrow - *Exception*; rectangle - *Domain Object*; diamond arrow - *Step*; solid line/triangle - *Subtype/Overriding*; dashed line/triangle - *Realization of a specification by one enactment*.

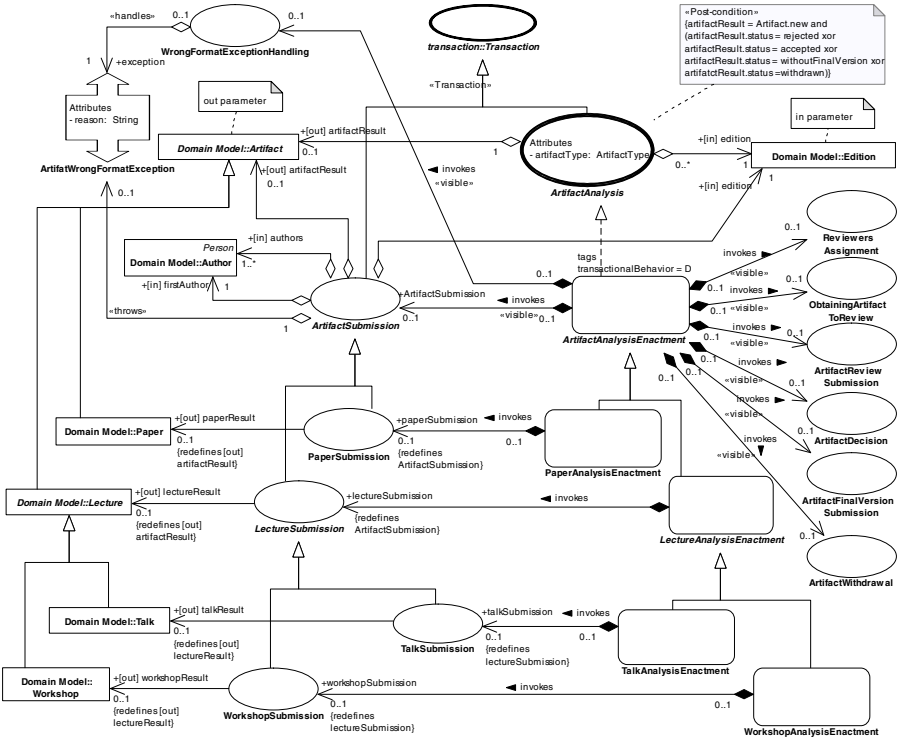

Fig. 3. Artifact Analysis business transaction informational model

The *ArtifactAnalysisEnactment* is composed (composition diamond symbol) by all Web transactions mentioned above (whose enactments are not shown). An interesting aspect is the overriding of *Artifact AnalysisEnactment* for each subtype of Artifact. This overriding is necessary because the *ArtifactSubmission* Web transaction is different for each subtype and, thus, the relationship with *ArtifactSubmission* must be redefined.

Similarly, the relationship with *Artifact* via the output parameter *artifactResult* is also refined for each subtype of *ArtifatctSubmission*. Other remarkable point is the abstract refinement from *ArtifactAnalysis* to *ArtifactSubmission*. At the *ArtifactAnalysis* level of abstraction there are only two parameters (*artifactResult and edition*) but at the level of *ArtifactSubmission* (and its subtypes) there are two extra parameters (*authors* and *firstAutor*). If the artifact is in the wrong format, the *ArtifactSubmission* Web transaction throws the *ArtifatWrongFormatException* that is handled by the Web transaction *WrongFormatExceptionHandling*, where the artifact is rejected.

Assuming that this conference works with multiple pc-chairs, we have to avoid concurrence interferences when one pc-chair allocates reviewers to an artifact (*ReviewersAssignment* Web transaction). Specifying that this Web transaction enactment ACID behavior has at least "I" which means that this Web transaction must be isolated elegantly solves this issue. This implies some lock implementation mechanism

in the running application, optimistic or pessimistic. Theoretically, "I" means *serializable*, but, in practice, this can be relaxed to augment concurrency and one may simply specify the level of isolation: *read uncommitted*, *read committed*, *repeatable read* and *serializable*. In this particular Web transaction, for example, we may use *repeatable read*, indicating that this Web transaction will have success only if no other concurrency session updates the artifact.

To complete the modeling, Figure 4[2] shows the flow model for the *ArtifactAnalysis* business transaction that defines execution flow of the transaction steps. Note that there are two scopes: one protected scope and one suspension scope. The first catches the *ArtifatWrongFormatException* thrown by the *ArtifactSubmission* step and sends to the *WrongFormatExceptionHandling* step. The other scope monitors the arrival of the external *WithdrawalEvent* that is generated by the user (author) when he wishes to withdraw the artifact. The rest is the traditional flow modeling. It is important to say that for each Artifact subtype (Paper, Workshop, and Talk) the corresponding *ArtifactSubmission* Web transaction subtype substitutes the *ArtifactSubmission* step in the flow model.

Finally, to illustrate the interplay between the transactional and navigational concerns, imagine that the designer wishes to permit the reviewer, when he is executing the *ArtifactReviewSubmission* Web transaction, to navigate to other artifacts that he has reviewed earlier. To achieve this, we have three independent models: the transaction model, where the *ArtifactReviewSubmission* Web transaction is specified, the navigational model, where the Artifact navigational node is, and an interface model, where there is an element that, when activated, sends two events, one to start *ArtifactReviewSubmission* Web transaction and another to change the navigation state to the Artifact navigational node. The resulting interface is the composition of the two executing "transaction" states: *ArtifactReviewSubmission* and navigation, which can be interpreted as a special Web transaction that never commits. In this way, the reviewer can navigate via the Artifact navigational node, suspending the *ArtifactReviewSubmission* Web transaction, which can be resumed later in the same form it has started.

At the end of this example, it becomes clear that following the roadmap suggested by the meta-model, the designer defines, explicitly, "all" properties and flow of each transactions, being forced to think about several important nuances that formerly were forgotten, and were only revealed in test (or production) phase, in the form of error (or defect).

In terms of software architecture, we can conjecture one transaction execution controller component - "Transaction Engine" - whose logic would already be defined by the models: "execution flow" - flow model pre-loaded; "(http) parameter cache" - reified transaction objects' attributes, and "concurrence control and lock management" - ACID properties of the transactions.

---

[2] DSL notation: rounded rectangle - *TriggerTransactionNode or CallOperationNode*; dashed rounded rectangle - *Scope*; arrow - *Edge*; diamond - *Decision/Merge Nodes*; bar - *Fork/Join Nodes*; solid circle - *InitialNode*; X circle - *FlowFinalNode*; solid/white circles - *FinalNode*; up arrow rectangle - *ThrowNode*; down arrow rectangle - *CatchNode*; pentagon - *SendEventNode*; chevron rectangle - *ReceiveEventNode*; little rectangle - *ObjectBufferNode*.

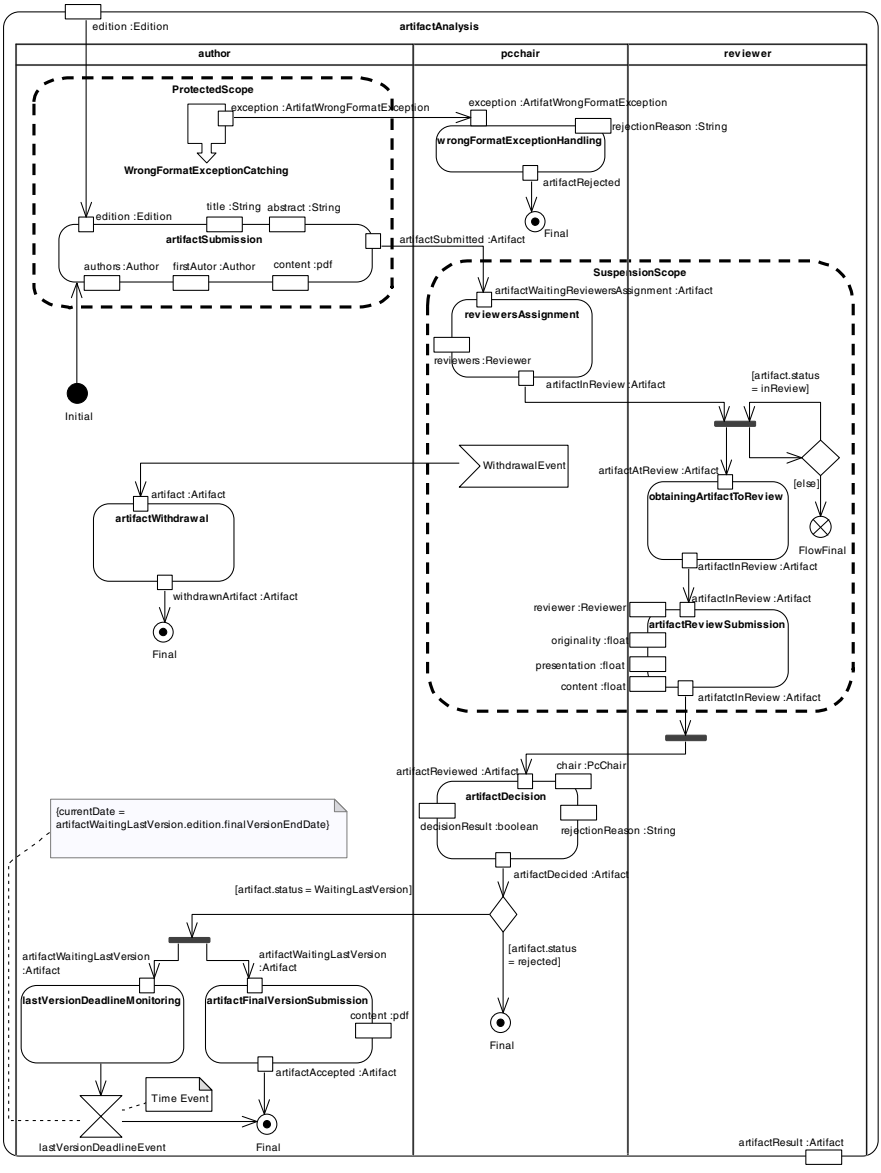

Fig. 4. Artifact Analysis business transaction behavioral model

4 Related Work

A number of well-known Web design methods have been extended to address transactions: Brambilla et all [2], OO-H and UWE [5], UWA [1], not detailed here for lack of space. None treat navigation as a special Web transaction, mixing up the navigational

and transactional concerns, with no clear separation among the specification and its various possible enactments. None of these proposals include all of the features present in our meta-model.

There are some transaction proposals for Web services (e.g. BPEL [8]) but these are for specifying Web services orchestration, based on Web services standards. For us, Web services are just one implementation choice of domain types operations.

5 Concluding Remarks

In this paper we have presented a reification approach to model business and Web transaction, covering the informational and behavioral perspectives of transactions. We have proposed a meta-model and graphical notation, defining a complete DSL to model transactions. We intend in future work to improve this DSL via many prototypes and to create other important Web application DSLs (e.g security DSL), which must be combined, without mixing the concerns involved.

Acknowledgement: This research was partially funded by CNPq, process number 142192/2007-4 and 302.352/85.6.

References

1. Baresi, L.: Ubiquitous Web applications. In: The eBusiness and eWork Conference (e2002), Prague, Czech Republic (2002)
2. Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I.: Process Modeling in Web Applications. ACM Transactions on Software Engineering and Methodology (TOSEM) 15, 360–409 (2006)
3. Distante, D., Tilley, S.: Conceptual Modeling of Web Application Transactions: Towards a Revised and Extended Version of the UWA Transaction Design Model. In: 11th International Multi-Media Modeling Conference (MMM 2005). IEEE Computer Society Press, Los Alamitos (2005)
4. Hee, K.V., Aalst, W.V.D.: Workflow Management: Models, Methods, and Systems. and Systems. MIT Press, Methods (2002)
5. Koch, N., Kraus, A., Cachero, C., Meliá, S.: Modeling Web Business Processes with OO-H and UWE. In: 3rd International Workshop on Web Oriented Software Technology (IWWOST 2003), Oviedo, Spain (2003)
6. Liskov, B., Guttag, J.: Program Development in Java: abstraction, specification, and object-oriented design, 6th printing. Addison-Wesley, Reading (2004)
7. Papazoglou, M.P.: Web Services and Business Transactions. World Wide Web: Internet and Web Information Systems 6(1), 49–91 (2003)
8. Web Services Business Process Execution Language, `http://docs.oasisopen.org/wsbpel/2.0/wsbpel-v2.0.pdf`