Engineering of Dependable Complex Business Processes using UML and Coordinated Atomic Actions

Nicolas Guelfi, Guillaume Le Cousin, Benoît Ries

Software Engineering Competence Center
Faculty of Science, Technology and Communication
University of Luxembourg, L-1359 Luxembourg-Kirchberg, Luxembourg
{nicolas.guelfi,guillaume.lecousin,benoit.ries}@uni.lu

Abstract. For many companies, it is widely recognized that languages and methods for modeling and analyzing distributed business processes are becoming more and more important. For improving efficiency, the modeling language should provide reusability, easy understanding by business analysts, and should ease the validation and verification tasks. In this paper, we present an approach for developing dependable complex business processes using UML that satisfies these requirements. The proposed UML notation is designed to be directly integrated with COALA, a syntactically and semantically well-defined fault-tolerant advanced transaction model based on Coordinated Atomic Actions. Structuring concepts like nested business processes and fault-tolerance through exception handling are first class concepts brought by our approach that are crucial for modeling cross-enterprise business processes. The modeling phase is followed by a validation phase by business analysts through animation of the business process model in a workflow environment. Due to the precise notation used, automatic verification of crucial properties is accessible through integration with an automatic verifier.

Keywords. Cross-enterprise business processes modeling, UML, advanced transaction model, fault-tolerance, validation, verification, methodology, tools.

1 Introduction

Business process modeling became a significant point for companies due to the growing need of competitiveness, responsiveness from the changes of market, but also for a better communication and comprehension of the businesses in an organization. Recently, many organizations cooperate forming partnerships to deliver solutions in a competitive way. They focus on their core business and outsource secondary activities to other organizations. Thus, companies require the ability to model business processes that are related to processes of their partners. As human dependence on technical infrastructure and systems grows, the threats of major failures grow accordingly. This fact calls for a concentrated effort to improve the

quality of Information and Communication technology (ICT)-based systems along the following axes: Security, Dependability and Trust. Concerning dependability, there are many application areas where system failure may lead to loss of financial resources, and even loss of human lives (e.g., control of aircrafts and nuclear plants). In such cases, scientifically rigorous evidence is needed in advance to back up promises about a product's future service. The problem is complicated by the need for practical ICT systems to evolve in response to changes in their requirements, technology and environments, without compromising their dependability. This is even stronger in applications where system boundaries are not fixed and are subject to constant urgent change (e.g. in e-business). Scientific and technological advances are required to (a) demonstrate that commercial and industrial-scale software can be developed to be truly dependable, and with less development risk than today; and (b) dependable systems can be evolved dependably including, for a class of applications, just-in-time creation of required services.

A business process model describes the activities that are carried out to reach a certain goal, and the execution order of these activities. It may involve many actors and perform many activities, which may be as simple as sending or receiving messages, or as complex as coordinating other processes or activities. An activity may also fail, which implies the integration of a rigorous fault-tolerant mechanism to deal with abnormal situations.

In order to represent such systems, the modeling language should describe actors with their role in the process, the activities execution flow (control flow) which may involve concurrency, and interactions between actors and between activities through messages exchange (data flow). The language should allow reusability of processes, and be easily understandable by non-computer specialists as business analysts. It must also have a well-defined semantics facilitating the validation and verification tasks, and thus ensuring to the involved organizations that the model is correct and fulfills the required properties.

We selected the Unified Modeling Language (UML) as the basis for our modeling language. Amongst all the UML [1] diagrams available, we consider that the UML activity diagrams syntax is well adapted for the behavior specification because it allows representing processes with actors, roles, activities execution flow and data flow in a graphic and comprehensive way [2], [3]. Activity diagrams syntax has been shown to be useful for business process modeling in [4]. UML Class diagrams represent the structure of data handled by the business processes. Moreover, UML provides extension facilities (stereotypes, tagged values and constraints) that allow building UML profiles for specific application domains, and it allows interoperability through the standardized open-source XML Metadata Interchange (XMI) format. Extensible Markup Language (XML) is a simple, very flexible text format, originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. Thus, in this paper, we define the syntax of our modeling language, which is a customization of UML activity and class diagrams.

The semantics is given using an automatic transformation of our UML models into a COALA model. The correspondence of formal semantics between UML and COALA is not an issue in this paper. The formal semantics is provided by the COALA model, thus business analysts must know that only the COALA models has

well-defined meaning, and thus only the COALA models, but not the UML models, should be used, and modified if necessary, for the final system validation. The UML models and the transformation from UML models to COALA models ease the formal specification of transactions to business analysts by modeling in a visual and more intelligible way for non-experts. COALA is a textual modeling language based on the Coordinated Atomic Action (CAA) concept [5], [6]. CAA provides a conceptual framework for dealing with cooperative or competitive concurrent processes. This is done by integrating three concepts:

- Conversations: provide cooperative activities, and is used to implement coordinated and disciplined error recovery;
- Nested transactions: maintain the consistency of shared resources in presence of failures and competitive concurrency, and allow a hierarchical transactions structure;
- Exception handling for error recovery.

Using the CAA concepts for modeling business processes allow cooperation between business partners, nesting transactions for a better reusability, fault-tolerance by using a transaction system [7], [8] for accessing shared resources, and exception handling for reacting as well as possible to an abnormal situation. The COALA language [9] provides a well-defined semantics of CAA and our approach aims at providing to COALA a UML-based syntax adapted to business process modeling, and suitable for verification and validation.

As for now, the modeling phase is followed by a validation phase handled by business analysts. During this phase, the models are transformed into XML Processing Description Language (XPDL) standardized format that is computed by a XPDL-compliant workflow engine. Business experts simulate the system behavior with the help of the workflow engine; i.e. they cooperatively validate the business model by playing roles of the business processes, and participate in the execution of the processes by receiving messages and sending answers. Since the validation cannot prove the absence of errors, but only detect some of them, we suggest to improve the current way of validation, by introducing an additional and complementary approach based on formal proofs and on the formal semantics of COALA given in terms of COOPN/2 (Concurrent Object-Oriented Petri Nets) expressions, allowing the verification of properties for all possible scenarios. We focus our work on the specification process, however an alternative approach for validation could be to use an execution platform that handles middleware implementations of CAAs like the DRIP platform [10]. This alternative approach differs by focusing on the generation of an executable prototype of the whole system.

This paper is structured as follows. Section 2 describes the coordinated atomic actions and COALA concepts, which are used to build our notation. This notation is described in Section 3. Section 4 describes the development process used for engineering complex business processes. Section 5 concludes and presents future work.

2 Coordinated Atomic Actions and COALA

The CAA concept [5], [6] facilitates the design of reliable distributed systems. It allows several threads (or *roles*) to perform a set of operations cooperatively using the conversation concept [11], [12] through *internal objects*. CAAs may be nested. Several nested CAA may be started concurrently and may be in competition for shared resources. To maintain the consistency of these resources in the presence of failures and competitive concurrency, the nested transaction model [13] is used. Moreover, CAAs integrate exception handling, in order to perform coordinated error recovery involving all the threads, or to propagate an exception to the enclosing CAA (possibly after having undone all its effects through the transaction system). The COALA language [9] provides a well-defined semantics to CAA in order to carry out dynamic properties verifications.

Fig. 1, below, illustrates an enclosing CAA involving three roles of which two carry out a nested CAA. This section will detail progressively the elements of the figure.

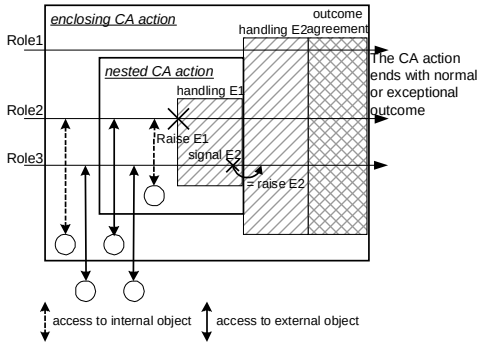

Fig. 1. Coordinated Atomic Actions

2.1 Fundamental Concepts of CAA

Roles. A CAA is composed of one or more *roles* (Fig. 1 shows three roles) executed in parallel to perform cooperatively a transaction fulfilling the ACID properties. The ACID model is one of the oldest and most important concepts of database theory. It sets forward four goals that every database management system must strive to achieve: Atomicity, Consistency, Isolation and Durability. The goal of a CAA consists in coordinating its roles. Each thread that wants to participate in a CAA takes a role by activating it. The CAA cannot start until all the roles are activated, i.e. roles start together.

Objects. The CAA concept defines two kinds of objects. *Internal objects* (shown as circles inside the CAA in Fig. 1) are internal to the CAA and are used to support

cooperative concurrency allowing the roles to agree upon the set of operations they wish to perform on the external objects. *External objects* (shown as circles outside of the CAA in Fig. 1) may be accessed by other CAAs executed concurrently, by using a transaction system ensuring the ACID properties.

Outcomes. All the roles exit a CAA at the same time, after each role has voted for its outcome. The four possible outcomes are:

- *Normal*. Indicates that the CAA was terminated and committed its results correctly while satisfying the ACID properties during execution.
- *Exceptional*. Indicates that the ACID properties were satisfied, but an exception was signaled to the enclosing CAA.
- *Abort*. Indicates that the CAA has aborted and undone all its operations while satisfying the ACID properties
- *Failure*. Indicates that an error occurred and the ACID properties could not be satisfied.

Nesting. A CAA can be nested into another one (as shown in Fig. 1). According to the nested transactions model, the effects of a CAA only become permanent when the top level enclosing CAA terminates. Nevertheless the effects of a nested CAA are visible to its enclosing CAA as soon as the nested CAA terminates.

Exceptions and Handlers. A CAA may contain two kinds of exceptions: *internal* and *interface* exceptions. In Fig. 1, exception E1 is internal to the nested CAA, and E2 is an interface exception of the nested CAA while it is internal to the enclosing CAA in order to handle it.

Internal exceptions are totally managed in the CAA, whereas interface exceptions correspond to the list of exceptions that the CAA may signal to its enclosing CAA (E2 is propagated to the enclosing CAA that handles it). By default, two interface exceptions exist: *abort* and *fail* (corresponding respectively to the abort and failure outcomes).

Due to the concurrent nature of the targeted systems, several exceptions may occur at the same time, and an exception requires all the roles to perform the recovery actions cooperatively. CAAs apply the following rules to handle exceptions:

- If internal exceptions are raised concurrently, an exception graph [14] is used to determine which handler to activate.
- When an internal exception is raised, all the roles activate the corresponding handler. If roles have entered a nested CAA, the resolution mechanism waits until the nested CAAs terminate (blocking method).
- If an interface exception is signaled by at least one role, all the roles signal the exception. If different interface exceptions are signaled, the CAA attempts to abort, if it fails the fail exception is signaled.
- If an interface exception is signaled while another role raises an internal exception, the raised exception is ignored.

2.2 COALA Language

COALA (COordinated atomic Action LAnguage) [9] is a formal language used for the specification of CAAs. A CAA is defined within a COALA module, which comprises two sections:

- An *interface* section, which is visible to other CAAs. It declares the list of roles with their parameters (the external objects), and the list of interface exceptions with their parameters;
- A *body* section, which is private and hidden to other CAAs. This section contains:

- The list of internal objects,
- The list of parameterized internal exceptions,
- A resolution graph which lists the combinations of internal exceptions that can be raised concurrently together with the resolved exception,
- A description of each role, consisting in the roles' behavior, the declaration of which handler must be activated for each resolved exception, and the behavior of each handler.

COALA provides additional features compared to the ones provided by the basic CAA concepts. A role is allowed to create new threads; however, a thread must terminate in the CAA where it was created. A role may be *asynchronous*, i.e. the role may enter the CAA during its execution and the CAA is allowed to start before the role is activated, however all the roles must synchronize at the end of the CAA and leave together. A role may require to be activated more than one time, possibly an indefinite number of times for asynchronous roles.

2.3 COALA's Semantics

COALA's semantics is defined as a translation from COALA programs into their formal description in COOPN/2 [15], which is an object-oriented specification language, based on Petri nets and algebraic data types. Three generic COOPN/2 classes are defined (*Caa*, *Role* and *Scheduler*), specifying the general behavior of CAA, roles, and access to external objects. The translation consists in generating a COOPN/2 class for each CAA and each role of the COALA program. Each of these classes inherits from one of the basic abstract classes (*Caa* or *Role*) and defines new axioms to specify their behavior.

3 FTT-UML: A UML-Profile for Fault-Tolerant Transactions

The proposed business process modeling language is based on the UML activity and class diagrams syntax and designed for automatic transformation into COALA, i.e. the modeling of COALA concepts, described in Section 2, with UML are eased with our notation: FTT-UML. We define our notation as a UML profile, i.e a customization and extension of existing UML elements. The syntactic elements used are *swimlanes* representing the participants of a business process (also called *roles*), *nodes* representing activities performed by a participant or representing exchanged

data, and *edges* representing the execution flow (solid lines) or the data flow (dotted lines). In the following sections, the flows will be specified in terms of token flow through the nodes. A node has pre- and post-condition, which may be the followings:

- Pre-condition XOR-merge: the node is executed for each incoming token
- Pre-condition AND-join: the node is executed only when a token is present for all its incoming edges
- Post-condition XOR-choice: after execution, a token is offered to only one outgoing edge
- Post-condition AND-fork: after execution, a token is offered to all the outgoing edges

3.1 Running example

To illustrate our notation, we will use a simple running example (see Fig. 2) involving several organizations, namely a customer, a wholesaler and a manufacturer; the management of abnormal situations as payment failure or production failure; and nested processes, as shown in Fig. 3.

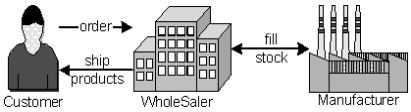

Fig. 2. Running example

When a customer sends an order to the wholesaler, the wholesaler launches a nested process *OrderTreatment* (see Fig. 3) consisting in two concurrent sets of activities for the wholesaler:

- Calculate the price of the order and launch the nested process *Payment* for the payment of the customer to the wholesaler;
- Check the stock. In case of an insufficient stock it sends a production order to the manufacturer and launches the nested process *Production*.

Two failures may occur:

- Production failure, if the manufacturer is unable to produce ordered products;
- Payment failure, if the customer is unable to pay the wholesaler. If the production fails, the wholesaler needs to abort the payment.

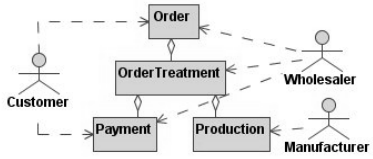

Fig. 3. Nested processes

3.2 Roles

A role in a business process represents a participating organizational unit. For satisfying the reusability requirement, a role has to be independent of the context where the process is executed, i.e. a role does not represent a physical organization, but the role that an organization has to take in the process (for example, the role *wholesaler* may be taken by different physical companies in different contexts).

Roles are represented by swimlanes, which are vertical lines dividing an activity diagram, as shown in Fig. 4. Each swimlane contains the activities performed by the corresponding role. From the COALA concepts, the execution of a process starts when all its roles are activated, excepted for asynchronous roles (represented by a stereotyped swimlane «asynch»), which may be activated during the execution of the process (for example, we could imagine that during the production of the manufacturer, an expert comes to perform verifications after a product was detected with a missing piece). However all the roles must synchronize at the end of the process and leave together with the same outcome.

The designer can specify that a synchronous role must be activated *n* times before the process starts, by adding <*n*> at the end of the role's name, or that an asynchronous role can be activated for an indefinite number of times by adding <*> (e.g. if the wholesaler performs an invitation to tender for the production, an indefinite number of manufacturers may join the process to make propositions).

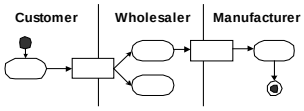

Fig. 4. Roles' notation

3.3 Data

Data structures are modeled using class diagrams (Fig. 5) and the corresponding data are used in activity diagrams using object nodes (Fig. 6) represented by rectangles. Each time a data structure is produced (a token comes in an object node: XOR-merge), a copy of this set of data is made available to all the activities that need it (a token is offered to all the outgoing edges of the object node: AND-fork). For example, each order sent by a customer will be transmitted to all the wholesaler's departments that need it.

Fig. 5. Class diagram modeling data structures

Fig. 6, below, illustrates data operations. In this figure, we model an agreement between the customer and the wholesaler: the customer sends an order, the wholesaler asks the price for the delivery and the production, then sends back the total price to the customer. At this moment, the customer can choose to make a new order replacing the old one, accept the order, or cancel the process. The process provides as output the order accepted by the customer.

In order to provide data manipulation primitives, the notation includes the following operations:

- Assign an expression to an object (or data structure) or an attribute. This is specified by an edge going to an object node. This edge has the stereotype «assign» and is labeled either with *expression* or *attribute=expression*, respectively to assign an expression to the object or an attribute of the object. For example, in Fig. 6 for assigning a value to the *total price* object.
- Copy the content of an object or an attribute into another object of same type, specified by an edge from the source object to an action node, and an edge from the action node to the target object. For copying an object, the latter edge has the stereotype «copy-of» and is labeled by the name of the source object, while for extracting an attribute the edge has the stereotype «extract» and its label has the format *object.attribute*. The copy is performed at the end of the action node execution. For example, in Fig. 6 for extracting the delivery address and the products from the order.
- Replace the content of an object that is not used yet. If a token still exists in an outgoing edge of the object node while a new one is coming, instead of queuing the new token, it replaces the existing one. This is specified by adding the stereotype «replace». For example to replace the *order* object for the action *agreement done* waiting the customer agreement (Fig. 6).

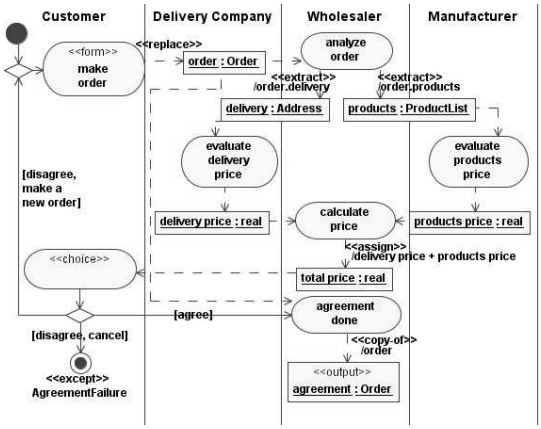

Fig. 6. Order agreement

3.4 Actions

A role performs actions, represented by action nodes (corner-rounded rectangles). An action is executed when all its pre-conditions are satisfied (AND-join), and offers a token to all its outgoing edges (AND-fork). For example, in Fig. 6 the action *calculate price* is executed once the delivery price and the products price are available. An action without specific stereotype represents a business activity, however the two following stereotypes may be used:

- «form» for information supplying. Incoming objects represent data giving details about the information to supply, while outgoing objects are supplied information. For example, in the action *make order*, the customer fills in the information related to his order.
- «choice» for taking a decision. Incoming objects represent data giving details about the decision to take. A decision node having guards representing possible choices must immediately follow such an action node. For example in Fig. 6, the customer uses the *total price* to choose either if he is *agree* or *disagree*. Stereotypes «form» and «choice» may be combined.

A diamond represents either a decision node if its outgoing edges have guards (in Fig. 6 the diamond under the action «choice») or a merge node if not (the diamond under the black dot). A decision node performs an AND-join / XOR-choice, while a merge node performs a XOR-merge / AND-fork.

3.5 Starting point and inputs

A process may take data as input. Object nodes having the stereotype «input» represent such data. The starting point node is either an initial node (represented by a black dot) or the only node depending only on «input» nodes (i.e. a node having only incoming edges from object nodes «input»). There must be a unique starting point amongst all the synchronous roles, which receives a token when the process starts. The asynchronous roles may have an initial node, which receives a token when the role enters the process.

3.6 Outcomes

A process terminates with a normal outcome if its objective is reached, or signal an exception to the enclosing process if a problem occurs. Moreover a process may produce outputs. A normal outcome is represented either by a final node (a bull's eye) if the process does not produce outputs, or by object nodes with stereotype «output». In the same way, an exceptional outcome is represented either by a final node with stereotype «except» and labeled with the exception's name, or by an object node with stereotype «except» named by the exception's name and representing the exception parameter. Fig. 6 shows a normal outcome with an output (agreement:Order) and an exceptional outcome without parameter (AgreementFailure).

3.7 Nested processes

A sub-activity node (represented by an action node with an icon in the lower right corner) models a nested process. A nested process being able to involve several roles, the roles of the enclosing process have to specify which role they want to take in the nested process. There are two possibilities for that:

- In the enclosing process, a role participating to the nested process has an edge going to the sub-activity node and labeled with the name of the role to take (Fig. 7 shows the *Wholesaler* taking the role *Paid* in *Payment*). If the role in the enclosing process does not contain the sub-activity node, it has an action node with the stereotype «invited» in its swimlane (Fig. 7 shows the *Customer* taking the role *Payer* in *Payment*).
- If the name of the role is the same in the enclosing process as in the nested process, and there is no edge to the sub-activity node labeled with the role's name, then it is considered that the role of the enclosing process takes the same role in the nested process. This in order to have clearer diagrams.

Providing inputs to a nested process is modeled by object nodes having an edge to the sub-activity node (object *price* for the process *Payment* in Fig. 7). However, if the process has several inputs having the same type, there is an ambiguity to know which incoming object corresponds to which input. To remove this ambiguity, the edge from an object to the sub-activity may have the stereotype «input» and be labeled with the name of the corresponding nested process' input.

After the nested process was terminated, the execution flow follows the outgoing edges corresponding to the outcome: edges having the stereotype «except» and labeled by the name of the signaled exception or edges without the stereotype «except» for a normal outcome. Such edges go into object nodes if the exception has a parameter or if the nested process produces outputs (in Fig. 7, *Production* may have a normal outcome producing *Products* or signal an exception *Abort* with a parameter of type *Notification*). To remove the ambiguity when several outputs have the same type, the edges may have the stereotype «output» and be labeled with the name of the nested process' outputs.

Undone the effects of a nested process is modeled by an edge with the stereotype «compensate» going to the nested process' sub-activity node (in Fig. 7 the edge from the object *failure* to the sub-activity *Payment* indicates that if the nested process *Production* fails the effects of the nested process *Payment* must be undone). If the nested process is running, it interrupts its activities, undoes all its effects and signals the *Abort* exception. If the nested process is already terminated, there are two possibilities:

- If it succeed, a compensating nested process (modeled by an activity diagram having the stereotype «compensate») is used to obliterate its effects.
- If it signaled the *Abort* exception, nothing is done.

Then the execution flow continues on the outgoing edge corresponding to the *Abort* exception.

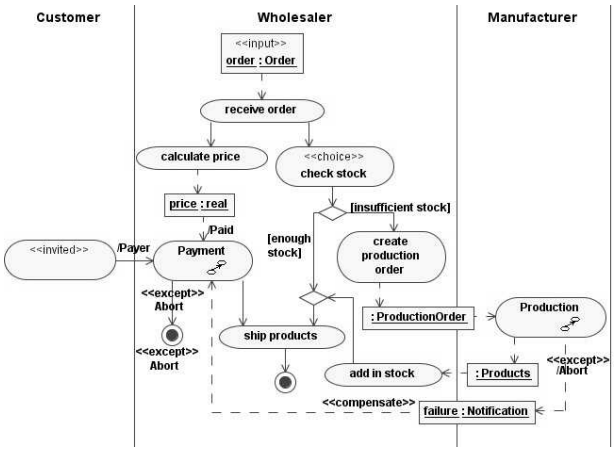

Fig. 7. OrderTreatment process

3.8 Internal exceptions

As a CAA's role, a role of a business process may raise an internal exception by using an action node with stereotype «raise» and named by the exception to raise. Such a node must not have outgoing edges. For resolving concurrent exceptions into a single exception to raise, an exception graph may be specified within a swimlane having the stereotype «ExceptionResolution». Such a swimlane contains an oriented graph, where action nodes represent exceptions, as shown in Fig. 8. A role may have a handler for each resolved exception, which is a sub-set of the role's activities starting from an action node with the stereotype «handler» and named by the exception's name. For a parameterized exception, the action «raise» has an incoming edge from an object node and the action «handler» has an outgoing edge to an object node representing the parameter.

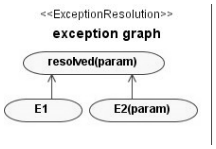

Fig. 8. Exception graph

4 Validating and Verifying Business Process Models

Our work is part of the E-fficient project, which aims at providing the business experts with a tool set for the modeling and the validation of e-business transactions.

However, we believe that our approach may be integrated into other methods involving business process modeling and validation. The first phase of an E-fficient project consists in the identification by business analysts of the need for e-business transactions. These needs arise from business opportunities that bring some added value to one or more organizations. These needs are represented within UML use case diagrams together with a class diagram representing the structure and the relations of the entities involved in the e-business transactions. Once the need of business processes has been clearly identified, their behavior is specified using our notation.

The modeling phase is followed by a validation phase by business analysts through animation of the business process model in a workflow environment. This animation is performed using a translation of the activity diagrams into a XPDL file [16] and class diagrams into XML Schemas [17]. These files are then used by a workflow engine, which allows business experts to cooperatively validate the business process model. Each expert plays one or more roles in the processes and participates in the execution of the processes by receiving messages and sending answers. Thus, business experts validate the processes by playing different possible scenarios. Validating a specification by animating it, is widely recognized to be a useful technique for business experts. But the successful application of this technique depends on the relevance of the chosen scenarios.

The validation by animation allows detecting errors and inconsistencies, but it cannot prove their absences. We suggest to complement this technique with an approach based on formal proofs that allows verifying a property for all possible scenarios. The properties to verify may be exhibited from errors discovered by business experts during the validation phase. [18] identifies four categories of properties to verify:

- Structural properties, which allow checking that diagrams are well formed according to the rules presented in section 3. This can be done by using the model checking tool USE to verify class and activity diagrams. The properties are specified with the OCL language.
- Static properties, which are related to a single state of the process.
- Dynamic properties, which are related to two or more states of the process. For verifying such properties together with static properties, we investigate the relevance of the use of COOPN. Actually, we perform an automatic transformation of business processes into a COALA program, providing its specification in COOPN language. The COOPN language is based on Petri nets and is provided with a prototyping tool for simulating the specification [19].
- Real time properties, which aim at evaluating the process in term of time response. Such properties are very useful in business processes since these processes are often time-critical (for example, a payment deadline or a maximum execution time for an activity or a process). [20] presents an approach for specifying real time constraints on an UML activity diagram by specializing the UML profile RT-UML [21], and for specifying real time properties by using a pseudo-english language translated into TCTL formulas [22]. This approach transforms an activity diagram into a timed automaton and use KRONOS tool for verifying properties expressed with TCTL formulas.

5 Conclusion

In this paper, we have presented an approach for modeling business processes using UML and CAA. We have shown how business analysts can develop business processes by using our extension to UML: FTT-UML. We have also shown that models done with our UML notation are specifically designed to be transformed into COALA models, which can then be verified with formal tools thanks to COALA's semantics based on the COOPN/2 formal specification language. CAA brings concepts for modeling cooperative and competitive concurrent processes with fault-tolerance, while UML provides a well-adapted syntax and interoperability through the standardized XMI format. We have defined an UML profile specializing activity diagrams for the business processes' behavior specification and class diagram for describing data structures.

The modeled business processes are validated by business experts through animation in a workflow environment and automatically transformed into a COALA program for future verification of static and dynamic properties.

Next step is to use COOPN/2 tools [23] to perform these verifications. The integration of our notation with [18] and [20] is also in progress in order to verify the four property categories: structural, static, dynamic and real-time properties. In [18] the verification of structural properties is based on the formal approach promoted by USE tool; in [20] KRONOS tool is used to verify temporal properties specified in pseudo-english.

BPMN [24] (Business Process Modeling Notation) has been proposed by BPMI and allows the generation of executable BPEL4WS [25]. This notation is very close to UML activity diagrams, and we believe that a UML profile provide the same capabilities. Moreover, BPMI and OMG plan to integrate the two notations in the future. [26] proposes a notation based on UML activity diagram for workflow modeling supported by a verification tool, but nested processes and exception handling are not integrated in this work. Our approach brings transactional and fault-tolerance capabilities through the CAA concepts, and aims at allowing validation and automatic verification.

6 References

[1] OMG, *Unified Modeling Language Specification Version 1.5*, 2003

[2] H.-E. Eriksson, M. Penker, *Business Modeling with UML: Business Patterns at Work*, John Wiley & Sons, 2000

[3] R. Eshuis, *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*, Ph.D. Thesis, University of Twente, 2002

[4] M. Dumas, A.H. Hofstede, *UML Activity Diagrams as a Workflow Specification Language*, 4th International Conference on the Unified Modeling Manguage, Modeling Languages, Concepts, and Tools, vol.2185, Toronto, Canada, Springer-Verlag, pp.76-90, 2001

[5] J. Xu, B. Randell, A. Romanovsky, C.M. Rubira et al. , *Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery*, Proceedings of the 25th International Symposium on Fault-Tolerant Computing, Pasadena, IEEE Computer Society, pp.499-508, 1995

[6] B. Randell, A. Romanovsky, R. Stroud, A. Zorzo, *Coordinated Atomic Actions: from Concept to Implementation*, Technical Report n°TR 595, Department of Computing, University of Newcastle upon Tyne, 1997
[7] P. Bernstein, V. Hadzilakos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987
[8] K. Ramamritham, P.K. Chrysanthis, *Advances in Concurrency Control and Transaction Processing*, Executive Briefing Serie, IEEE Computer Society Press, 1997
[9] J. Vachon, *COALA: A Design Language for Reliable Distributed Systems*, Ph.D. Thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, #2302, 2000
[10] A. Zorzo, R. Stroud, *An Object-Oriented Framework for Dependable Multiparty Interactions*, OOPSLA-99, pp.435-446, 1999
[11] B. Randell, *System Structure for Software Fault Tolerance*, IEEE Transactions on Software Engineering, vol.SE-1, pp.220-232, 1975
[12] B. Randell, J. Xu, *The Evolution of the Recovery Block Concept*, Software Fault Tolerance, pp. 1-21, 1995
[13] J. Moss, *Nested transactions: an approach to reliable distributed computing*, Massachusetts Institute of Technology, 1981
[14] R.H. Campbell, B. Randell, *Error recovery in asynchronous systems*, IEEE Transactions on Software Engineering, vol.12, IEEE Press, pp.811-826, 1986
[15] O. Biberstein, D. Buchs, N. Guelfi, *Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism*, Advances in Petri Nets on Object-Orientation, Lecture Notes in Computer Science, Springer-Verlag, 2001
[16] WfMC, *XML Processing Description Language*, http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf, 2002
[17] W3C, *XML Schema Type Formats*, http://www.w3.org/TR/xmlschema-2/,
[18] N. Guelfi, A. Mammar, B. Ries, *A Formal Approach for the Specification and the Verification of UML Structural Properties: Application to E-Business Domain*, Submitted to the 6th International Conference on Formal Engineering Methods (ICEFM), Seattle, WA, USA, 2004
[19] S. Chachkov, D. Buchs, *From Formal Specifications to Ready-to-Use Software Components: The Concurrent Object Oriented Petri Net Approach*, International Conference on Application of Concurrency to System Design, Newcastle, IEEE Press, pp.99-110, 2001
[20] R. Annonier, N. Guelfi, A. Mammar, *Verification of Real-Time e-Business Transactions using RT-UML and KRONOS*, Submitted to, 2004
[21] OMG, *UML Profile for Schedulability, Performance and Time*, Specification OMG, 2003
[22] R.A, C.C, D.L. Dill, *Model-Checking in Dense Real-Time*, Information and Computation 104(1), pp.2-34, 1993
[23] C. Péraire, S. Barbey, D. Buchs, *Test Selection for Object-Oriented Software Based on Formal Specifications*, IFIP Working Conference on Programming Concepts and Methods (PROCOMET'98), Shelter Island, New York, USA, pp.385-403, 1998
[24] BPMI, *Business Process Modeling Notation (BPMN) 1.0*, BPMI.org, 2004
[25] Microsoft, IBM, Siebel, BEA, SAP, *Business Process Execution Language for Web Services*, 2003
[26] R. Eshuis, R. Wieringa, *Tool Support for Verifying UML Activity Diagrams*, vol.30, IEEE Press, pp.437-447, 2004