Extending the Consistency Property of Software Transactional Memory Systems with business constraints in OCL and incremental checking

Alberto-Manuel FERNANDEZ-ALVAREZ,
Daniel FERNANDEZ-LANVIN and
Manuel QUINTELA-PUMARES
Computing Department, University of Oviedo
Oviedo, Asturias, Spain

ABSTRACT

Software Transactional Memory (STM) systems offer AcId (note the letter case) properties. Consistency is usually regarded from the STM side but not from the business problem point of view. We propose a way to extend current STM systems with a consistency subsystem able to verify business constraints expressed in OCL by the development team. The proposed solution applies OCL incremental checking techniques and code generation to couple seamlessly with the transaction lifecycle with little runtime performance penalty.

Keywords: Constraints, Transactional Memory, Consistency checking, Incremental checking.

1. INTRODUCTION

Software Transactional Memory (STM)[1] systems bring the ACID properties to the mainstream programming: the well-known *Atomicity*, *Consistency*, *Isolation* and *Duration* properties. However, in STM, *Consistency* and *Duration* differ a bit their interpretation in respect with the database field. In STM, *Duration* is not understood as persistence, but as the visible effects after the transaction commit. On the other hand, *Consistency* is considered from the point of view of the concurrency management system itself, as the STM being able to serialize the inner operations of all concurrent transactions. That interpretation of *Consistency* lacks the business point of view.

For instance, constraints like "*a motorbike has two wheels*" cannot be easily expressed and checked as an integrity validation prior to the commit operation. However, this type of constraints, along with the domain model specification, is a very frequent analysis outcome. If developers were able to express those constraints, identified by the analyst, in a high level language, and got them checked automatically as the *Consistency* property of ACID transactions, they would produce higher quality software without increasing the effort or the complexity of the developing task.

The most widely used language in Object Oriented development to represent constraints is the Object Constraint Language (OCL)[2]. This language is used by analysts and developers, mostly by those that follow a MDE approach. The subsequent implementation of those constraints usually involves some difficulties that can complicate the work of the programmers: (1) *how* to write the invariant check, (2) *when* to execute the constraint verification, (3) *over-what-objects* should be executed and (4) *what-to-do* in case of a constraint violation.

Although all these issues could be solved by manual implementation, their complexity makes its development and maintenance an error prone task that implies a potential source of issues.

We think that all the aforementioned difficulties could be avoided by means of appropriate automated consistency checking mechanisms that complement STM systems.

2. DIFFICULTIES OF CONSTRAINT IMPLEMENTATION

Constraints that affect to just one attribute, or to a set of attributes on the same object, can be easily checked (the *how* problem) as invariants, or post-conditions, in a Design by Contract (DbC) way. However, those that affect more than one object of the domain (domain constraints) are determined by the state and relationships of every concerned object. As changes in the state of any of the involved objects can be produced by different method calls, it is difficult to know where to place the constraint checking code. According to the DbC approach, these invariants will only be checked if any public method of the invariant's declaring class object is executed, but modifications to the other involved objects will not be detected (this is a known limitation of DbC, referred as the framework problem [3]). The developer may then try to scatter the constraint along several methods, even in different classes. That will lead to code scattering, code tangling and tight coupling [4].

The second issue, the *when* question, is also problematic. In case of domain constraints, immediate checking after every single method call could simply not be possible, as low-level method calls may produce transient illegal states that, eventually, will evolve to a final legal state. That means the checking must be delayed until the higher-level method finishes. However, it may be difficult to foresee at programming time whether a high-level method is being called by another higher-level call or not.

The third issue (*over-what-objects*) developers must solve is to delimit the scope of the checking. A complete checking of every constraint after any modification would involve unfeasible performance rates. Ideally, the programmer must keep track of those constraints that might be compromised, and the affected objects, and then, at the end of the high-level method, check as few constraints, over as few objects, as possible.

Finally, developers must guarantee the consistence of the model in case a domain constraint violation happens (*what-to-do*). There are many works on this topic. Some applies backward error recovery techniques (BER) that provide *Atomicity*, that is the case of Reconstructors [5]. With that property in place, programmers can assume that modifications are done in an all-or-nothing way.

3. PROPOSED SOLUTION

Checking all pending constraints when the transaction is about to commit solves the *when* difficulty (2) and enriches the Consistency property of ACID transactions from the point of view the business.

What-to-do (4) in case of failure is then solved due to the Atomicity property of transactions.

The *how* (1) difficulty can be solved applying OCL analysis techniques and code generation. Those techniques are able to convert the original OCL constraints into new incremental versions optimized for the modification events that might occur at runtime to the object graph. These new optimized OCL versions are then translated into source code ready to be compiled with the rest of the application code.

Finally, the *over-what-object* (3) difficulty can be solved by generating code to detect the modifying events that might violate the constraints. These event detectors need the support of a runtime environment that stacks triples of event-object-constraint to check. Consequently, programmers' effort would be reduced and STM enriched with business Consistency.

4. CONSTRAINTS ANALYSIS PROCESS

The constraints analysis process is executed at compile time (Fig. 1).

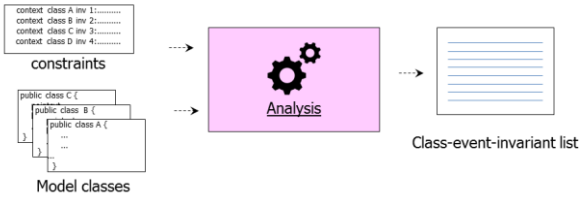

Fig. 1. Input and output of the analysis process.

The outcome of the analysis process is a relation of classes, events and invariants whose organization is shown in Fig. 2. For each object in the graph (an instance of a ModelClass) the process determines which events might violate which invariants that must then be checked under the context of (may be another, due to a context redefinition) object over which the invariant is defined.

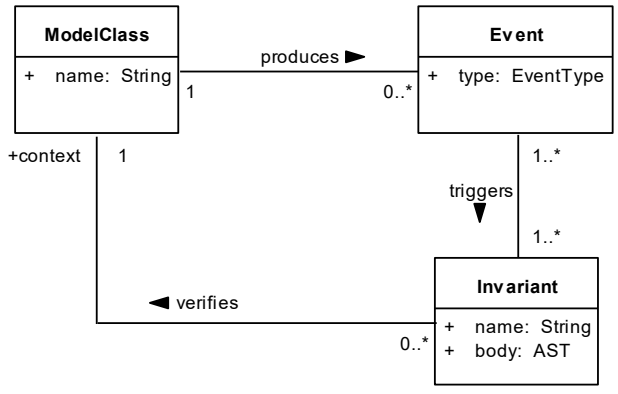

Fig. 2. Structure of the information produced by the analysis process.

With that information available, is easy to automate the generation of code for the event detection and the new versioned constraints.

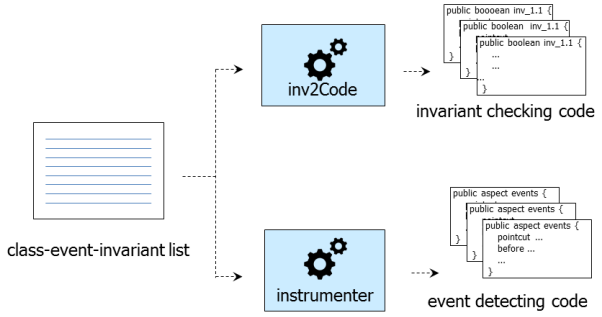

Fig. 3. The output of the analysis process is used to generate code for event detectors and checking code.

The processing of the constraint work over the abstract syntax tree (AST) and consists of several stages, as depicted in Fig. 4.

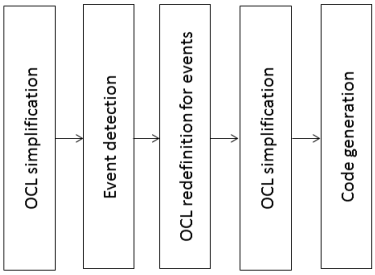

Fig. 4. Processing of constraints.

During the first stage OCL expressions are simplified by translating them into a canonical form. During this step some logical equivalences are applied. An extensive relation of these can be found in [6]. The process uses here a rule engine that recursively applies every matching rule over the AST until no more rules can be applied.

The second stage computes all possible structural events that can affect a constraint. To this end, we follow the process explained in [7]. Our implementation can detect five different types of events:

- *Insert*: the creation of a new entity (call to new operator)
- *Delete*: the deletion of an entity. There are some extra difficulties here as in Java we cannot delete an object. More on this later.
- *Link*: indicates the linking of two objects over an association.
- *Unlink*: signals the unlinking of two objects.
- *UpdAtt*: indicates a change in the value of an attribute (update attribute).

The third stage computes for each constraint-event pair a new alternative equivalent to the first but probably simpler and with fewer entities involved. This new constraint is specialized for that specific type of event.

After this transformation, the simplification rule engine is executed again with the addition of some new rules [7]. After this sequence of steps, we end up with some simpler constraints regarding every event for each constraint over a class. These new constraints will be simpler and intended to be checked only if its specific firing events occur.

5. IMPLEMENTATION ISSUES

This proposal can be applied over a wide variety of object-oriented programming languages. The most problematic issues are related to the OCL expressions processing, domain model loading and modifying events detection at runtime.

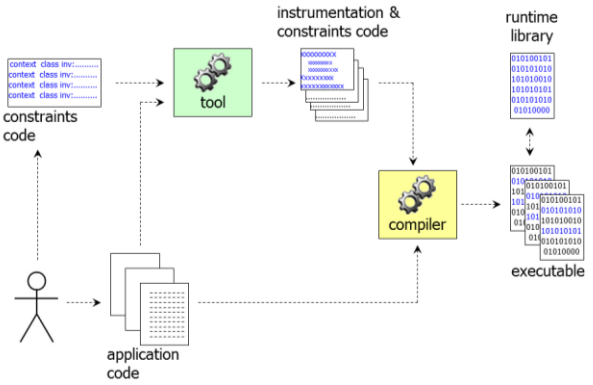

Fig. 5. Parsing OCL, loading the model, generating code and the support of a runtime.

For the first issues there are three alternatives: (i) to use an available OCL parser, (ii) to write one in the target language or (iii) to make a cross-processing (a constraints analyzer written in any language other than the development one).

For the second there are also several options: structural introspection (supported by most modern languages) of the classes implementing the domain model (e.g., the Java Reflection API), or the modelling of the domain model in any format the constraint analyzer can process (as produced by modeling tools, e.g., XMI).

After the analysis step, some code generation is required to instrument the detection of events and to stack *linking objects* in the context of the current transaction. That *linking objects* relate the detector object (the one that suffers the modification event) with the constraint that must be checked and the navigation path to travel from the detector object to the constraint context object. The supporting runtime must provide a way to store those *linking objects* until the commit of the transaction, providing some optimizations to avoid repeated linking objects. It is also needed to translate into executable code the new versions of the constraints.

The generation will be conditioned by the architecture of the target application. It is quite likely that the execution platform provides some aid that will simplify the task (frameworks, interceptors, aspect compiler, etc.). Even if the environment is simple (the bare language and its core libraries) it will probably have tools to apply aspect oriented programming, or to allow some type of bytecode manipulation. Otherwise, as a last resort, we could preprocess the source code created by developers.

The last issue is the integration of the supporting runtime in the transaction lifecycle, as all the checking must be done prior to commit operation. The supporting runtime will offer a context to stack the linking objects and finally execute the pending checkings. That context follows a resource usage pattern (create /open, use, close):

```
Context ctx = Context.createContext();
try {
   <business operations here>
   ctx.close();
} catch (…) {
   ctx.dispose();
}
```

Fortunately, the transactions have the same pattern. This feature allows nesting naturally:

*(create/open, **use***, close)*

Where ***use**** could expand to:

(create, execute, verify, close)

What results in:

(create/open, (create, execute, verify, close), close)

Being the transaction the outer pair of parentheses and the constraint checking context the inner one.

Therefore, the proposed idea seems to be easy to integrate with not only STM but other systems with a resource like patterns such as object-relational mappers [8] or Reconstructors [5]. The only requirement for this is that the target system we want to integrate it with must offer a registration point, or any other type of interception mechanism.

We think all these discussed alternatives open a range of possible languages, and target platforms broad enough to consider the proposal of wide application.

6. RELATED WORK

Simon Peyton Jones *et al.* [9] propose an extension to the STM library for Haskell [10] which allows the programmer to express invariants in the same language. These invariants are automatically verified before the commit, and, in case of violation a rollback restores the initial state.

The mechanism of checking the consistency is highly integrated with the control system of transactions. This STM is built around the concept of TVar, i.e. a transactional variable. A TVar is able to maintain the previous values for each concurrent thread acting as a redo-log. When the programmer adds an invariant to the functional code, the STM detects its dependencies on TVars. In that way, this system maintains a list of invariants to check if there are changes in the TVar. It is, therefore, a mechanism for incremental verification that only checks the necessary assertions according to the events produced.

Another interesting aspect of this implementation is the ability to add transitional (or dynamic) restrictions. As TVars store the value in memory before the modification (necessary for the rollback), it is possible to access the original value and the current one, and express constraints comparing the two states.

João Cachopo in his thesis proposes a similar mechanism to check constraints for his STM called JVSTM [4]. The paper proposes and implements a STM in Java, for use in production, based on VBoxes (versioned boxes)[11]. The idea is similar to the one described by Peyton Jones *et al*.

The restrictions are implemented in Java and are part of the business code. They are methods denoted by the @ConsistencyPredicate annotation, without parameters and return a boolean. The system, similarly to the Peyton Jones *et al*. proposal, records the dependencies between the assertion

methods and the VBoxes by monitoring the accesses from the assertion methods to the VBoxes when the object is first joined to the transaction.

Both systems share advantages and disadvantages. Both of then offer an efficient verification due to the recording of dependencies between TVar/VBox and the methods that must be checked. This is, in fact, a type of incremental checking as the proposed here. Both, also, offer support for all types of constraints. And in both cases the assertions are written with the same programing language, which might alleviate the learning curve for developers.

However, on the one hand, the two solutions are very dependent on the systems they are embedded in, while we offer a general solution that might be integrated with other deployments (e.g., ORM, plain Java, Reconstructors, etc.). On the other hand, the way both detect the dependencies between TVar/VBox and the assertion methods is by monitoring the accesses done from the code while it is in execution. But these depend on the execution flow inside the method; that is, if the method has conditionals, switch statements or polymorphic invocations not all branches will be covered so there are chances to overlook some dependencies.

7. PROTOTYPE IMPLEMENTATION

In order to proof the feasibility of this proposal a prototype has been implemented. We chose the Java platform as the target as it is a mainstream language and offers a wide variety of tools and frameworks that easy the task. Multiverse [12] is used as the STM implementation. It is a STM implementation based on the use of atomic references (similar to the TVar of Haskell or the VBox of JVSTM). The transactional code must be executed under a template method (*atomic()*) and written so that it implements a *Txn<Type>Callable* interface:

```
public class Department {
   private TxnRef<String> name = newTxnRef<>();

   public void aBusinesMethod( … ) {
      atomic( new TxnVoidCallable() {
         public void call(Txn trx) {
           <business code here>
      });
   }
   ...
}
```

As OCL parser we use the Dresden OCL ToolKit (DOT) [13]. It provides a set of tools to parse and evaluate OCL constraints on various models like UML, EMF and Java, and is also able to generate Java equivalent code. We take advantage of the model loading feature to build a representation of the domain model from Java classes. The OCL parsing capabilities are also used to build an AST representation of every OCL constraint.

After obtaining an AST for each constraint, the analyzer, implementing the algorithms presented in the section 4, produces the resulting information we feed into a code generator that yields the instrumenting code for event detection and the implementation of the new-versioned constraints. Both the instrumenting code and the constraint checking code are woven with developer's code by using aspect orientation. In our prototype the code generator produces AspectJ code [14].

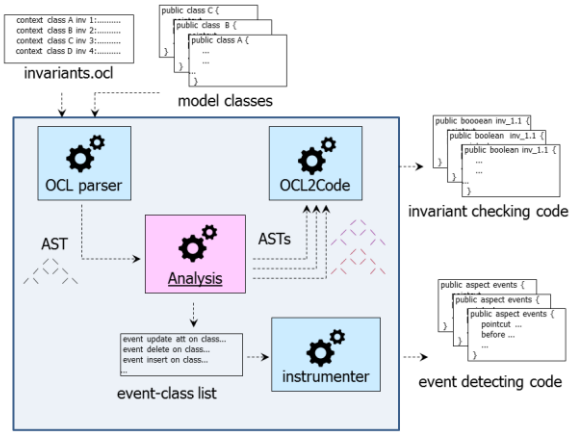

Fig. 6. Structure of the prototype implementation.

The code produced for event detection has to deal with these different situations: (i) detect modifications of attributes, (ii) detect the linkage (also the opposite) of objects and (iii) detect creation and deletion of objects.

The first (i) is solved by issuing an *after advice*[1] on every attribute of a domain model class. That advice is in charge of creating the *linking object* and staking it onto the constraint context.

The second (ii) is more convoluted. As UML associations are implemented with object references, establishing an association between two objects implies to modify one or two references (in case of a bidirectional association). For the sides with maximum multiplicity of one, the case is similar to the modification of an attribute. But for a many side, usually implemented as an attribute of a collection type, creating/deleting a link means to insert or remove to/from that collection. For that case, our prototype inserts a proxy to intercept these operations and then generate and stack the *linking object*. The proxy is set with an *after advice* that watches the first assignation of the collection attribute.

The third situation (iii) has to deal with the creation and deletion of objects. Just detecting the construction of an object (with an *after advice* on the constructor execution) will not be enough. Some objects could be created just as temporal values (variables in methods) and others could be unreferenced objects waiting for the garbage collector to be removed. Therefore, several questions come to the fore here: Which objects are valid objects? When does an object become invalid (i.e. it is no longer used)? Where is the collection of valid objects?

In our understanding, the objects that must be considered valid are those in the domain object graph, more precisely, those objects reachable from the graph. In that way, we can detect the addition of a new object when it is linked to another object already present in the graph. Conversely, an object deletion will be produced after the removal of all links that maintain the object linked to the graph. In that way, we consider an object to be *in-the-graph* when it is reachable though "any" link of "any" association type its class can have. Using another aspect, we crosscut the domain entity classes with two collections, one for forward references, and another for the backward ones.

A more precise definition of these aspects can be found on a previous paper of the authors [15].

[1] AspectJ reserved words.

The supporting runtime must provide a context able to stack the linking objects and then verify all the pending checking at the *close()* operation.

Embedding the constraint context in the transaction context
The last issue is to link both contexts so that, when the transaction is about to commit, all the pending checking will be executed. In this prototype we use, again, aspects to intercept the transaction. The atomic static method provided by Multiverse is schematized as follows:

```
public void atomic(
     final TxnVoidCallable callable){

   Trx tx = txnFactory.newTransaction(…);
   boolean abort = true;
   try {
      callable.call(tx);
      tx.commit();
   } catch (Exception e) {
      ...
      abort = true;
   } finally {
       if (abort) tx.abort();
   }
   ...
}
```

The solution adopted is to intercept the execution of the *callable.call()* method from the *atomic()* method and then handle the constraint checking context properly. This can be done with a simple aspect like the one shown below:

```
public privileged aspect InvariantInterceptor {
   pointcut transactionalCallables() :
     execution(
      * org.multiverse.api.callables.*.call(Txn)
     );

   Object around(): transactionalCallables() {
      Object res = null;
      Context ctx = Context.createContext();
      try {
         res = proceed(); // <-- callable.call()
         ctx.close();  // <-- invariant checking
      } catch (Exception ex) {
         ctx.dispose();
         throw ex;
      }
      return res;
   }
}
```

8. EVALUATION

Although the basic idea of this proposal (to add business consistency to an STM system) is validated with the construction of a working prototype, we have designed two experiments to measure its runtime performance costs. The first one compares the benefit of incremental checking versus naïve checking (all constraints are checked against all objects in the domain graph). The second one compares the transaction times with and without incremental constraint checking.

We designed a small application for project management. The domain model (Fig. 7) consists of 5 entities with 7 bidirectional associations among them.

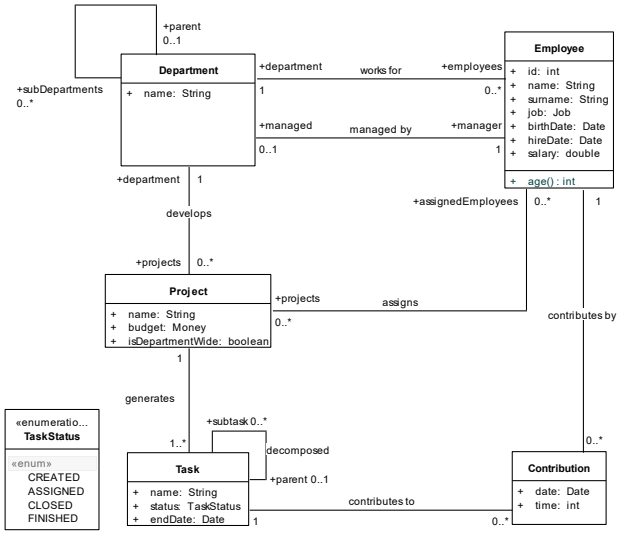

Fig. 7. Example domain model for the experiment.

The small application solves 8 use cases to cover the life cycle of a project. Along with use cases, 10 constraints are defined so that all types of constraints are covered. The use cases were designed in a way they produced events to check all the constraints. After the analysis done by the tool, 32 events were computed, some of them affecting more than one constraint, and 5 new event-specialized versions of constraints were generated.

For the first experiment, all use cases are run in sequence (let's call it a test run) and the total checking time is measured. Every use case in the sequence is run over the same preexisting graph of objects, recreated before each use case execution, and modifies the same number of objects. What is changed from one test run to another is the size of the initial graph. This is varied from 300 up to 20.000 objects.

The experiment was conducted with a JVM set to use 5 GB of heap size (to reduce the noise of garbage collection) and jittering enabled. Each test run was conducted 5000 times to heat the JVM and then the 90 best times of the next 100 execution times were taken.

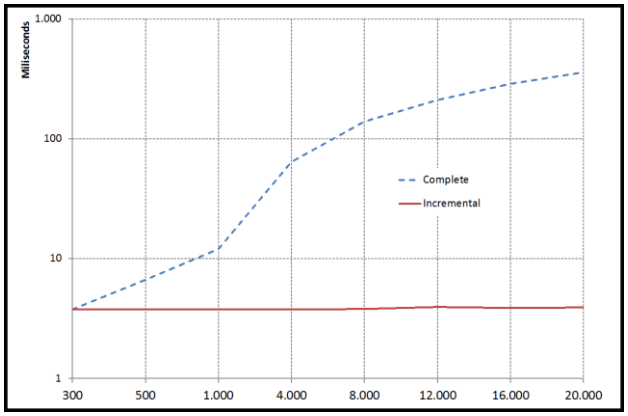

Fig. 8: Complete checking vs incremental checking times.

The results are shown in the Fig. 8. We can observe, as expected, the incremental execution times remains constant and independent of the graph size, while the complete checking time increases with the size of the object graph. At the minimum graph size both times are equal, as 300 is the number of objects

the incremental checking has to visit with our experiment design.

The second experiment tries to measure the extra time taken for consistency checking in the total transaction time. For this case, we set a fixed object graph size of 4000 objects (the previous experiment showed the checking time is independent from the object graph size) and measure the business execution and checking times for each use case. The result is shown in the Fig. 9.

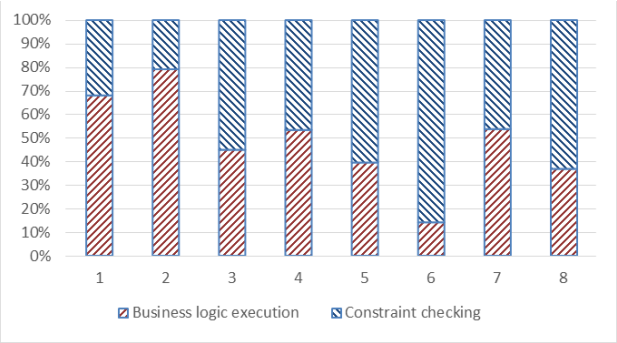

Fig. 9. Execution time vs checking time for each use case.

The graph shows different increments, ranging from 20% to 80%. That oscillation indicates that there is no structural deficiency on the checking system, as in that case all the ratios will be high. This difference comes from the fact that business and the checking operations can have different complexities, being independent from one another. A small modification in the object graph might raise one or more complex checking, while a complex business operation might affect just one simple-to-check constraint.

9. CONCLUSIONS

The proposed solution takes the best of STM and OCL worlds and benefits both. On the one hand, it solves the developers' difficulties to implement constraint checking mechanisms. The use of an STM easies the *when* and *what-to-do* difficulties of constraint implementation, while incremental OCL checking techniques and code generation solves the *how* and *over-what-objects* difficulties. On the other hand, it allows to enrich the *Consistency* property of current STM systems in a way independent of the concrete STM implementation.

As shown with the experiments that integration seems to be feasible and performs well, not introducing structural overloads in total transaction time.

10. ACKNOWLEDGEMENTS

This work has been funded by the European Union, through the European Regional Development Funds (ERDF); and the Principality of Asturias, through its Science, Technology and Innovation Plan (grant GRUPIN14-100).

11. REFERENCES

[1] T. Harris, J. Larus, and R. Rajwar, "Transactional Memory, 2nd edition," *Synthesis Lectures on Computer Architecture*, vol. 5. pp. 1–263, 2010.

[2] Omg, "OMG Object Constraint Language (OCL) v2.3.1," vol. 3, no. January, p. 246, 2012.

[3] B. Meyer, "Framing The Frame," *NATO Sci. Peace Secur.*, no. Series D: Information and Communication Security, pp. 174–185, 2015.

[4] J. M. P. Cachopo, "Development of Rich Domain Models with Atomic Actions," UNIVERSIDADE TÉCNICA DE LISBOA, 2007.

[5] D. Fernández Lanvin, R. Izquierdo Castanedo, A. A. Juan Fuente, and A. M. Fernández Álvarez, "Extending object-oriented languages with backward error recovery integrated support," *Comput. Lang. Syst. Struct.*, vol. 36, no. 2, pp. 123–141, Jul. 2010.

[6] J. Cabot and E. Teniente, "Transformation techniques for OCL constraints," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 179–195, Oct. 2007.

[7] J. Cabot and E. Teniente, "Incremental integrity checking of UML/OCL conceptual schemas," 2009.

[8] C. Bauer, G. King, and G. Gregory, *Java Persistence with Hibernate*. Manning Publications Co., 2014.

[9] S. P. Jones, "Transactional memory with data invariants," in *First ACM SIGPLAN Workshop on Languages Compilers and Hardware Support for Transactional Computing TRANSACT'06 Ottowa*, 2006.

[10] G. Korland, N. Shavit, and P. Felber, "Noninvasive concurrency with Java STM," *Work. Program. Issues Multi-Core Comput.*, pp. 7–20, 2010.

[11] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Sci. Comput. Program.*, vol. 63, pp. 172–185, 2006.

[12] "Multiverse : Software Transactional Memory for Java." [Online]. Available: http://multiverse.codehaus.org/overview.html.

[13] Claas Wilke, Dr.-Ing. Birgit Demuth, and Prof. Dr. rer. nat. habil. Uwe Aÿmann, "Java Code Generation for Dresden OCL2 for Eclipse." Feb-2009.

[14] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm, "AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts," in *Proceedings of the 13th International Conference on Modularity*, 2014, pp. 157–168.

[15] A.-M. Fernández-Álvarez, D. Fernández-Lanvin, and M. Quintela-Pumares, "Invariant Implementation for Domain Models Applying Incremental OCL Techniques," in *Software Technologies: 10th International Joint Conference, ICSOFT 2015, Colmar, France, July 20-22, 2015, Revised Selected Papers*, P. Lorenz, J. Cardoso, A. L. Maciaszek, and M. van Sinderen, Eds. Cham: Springer International Publishing, 2016, pp. 137–154.