OCL for Rich Domain Models Implementation
An Incremental Aspect based Solution

Alberto-Manuel Fernández-Álvarez, Daniel Fernández-Lanvin and Manuel Quintela-Pumares
Computing Science Department, University of Oviedo, Oviedo, Asturias, Spain

Keywords: Constraint, Invariant, Incremental Checking, Domain Model, Object Orientation, OCL, AspectJ.

Abstract: Object Constraint Language (OCL) can be used to express domain model constraints. Those related to a single object are easy to implement. However, when a constraint depends on the state of more than one object (domain and class constraints) the problem turns much more complicated. Developers must deal with several difficulties: how to write the invariant check, when to execute the constraint verification, over what objects and what to do in case of a constraint violation. Things are harder if we add feasible performance as requirement. We propose a tool that combines incremental OCL processing, with translation into aspect code and execution inside an atomicity execution context. The output is aspect code, ready to be integrated with business code that checks all the invariants efficiently at the end of the atomic operation.

1 INTRODUCTION

Domain modelling is a well-known practice to capture the essential semantic of a rich domain. The advantages of this approach have been widely discussed in the software engineering literature (Evans, 2003; Fowler, 2003; Olivé, 2005). The most popular tool to model both static and dynamic aspects of the domain model during the development is UML. Even though UML is proven as an effective and powerful resource, it is lacking in mechanisms to represent efficiently some aspects of the system under design. For instance, some complex domain constraints cannot be easily graphically expressed in UML.

Those constraints can be expressed in natural language or by means of OCL expressions that complement the UML models. Afterwards, the developer will transform these OCL expressions into source code.

Although the use of OCL fills the gap of UML limitations, the subsequent implementation of these 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.

Constraints that affect only to one attribute or set of attributes on the same object can be easily checked (*how*), as invariants or post-conditions in a Design by Contract (DbC) way. However, those that affect to more than one object of the domain (domain constraints) are determined by the state and relationships of every concerned object. As changes in any of the involved objects state can be produced by different method calls it is difficult to know where to place the constraint checking code. If we apply DbC, 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 (Meyer, 2015)). The developer may then try to scatter the constraint over several methods, even in different classes. That will lead to code scattering, code tangling and thigh coupling (Cachopo, 2007).

Regarding the "*when*" problem, 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. Again, 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

Fernández-Álvarez A., Fernández-Lanvin D. and Quintela-Pumares M..
OCL for Rich Domain Models Implementation - An Incremental Aspect based Solution.
DOI: 10.5220/0005517201210129
In *Proceedings of the 10th International Conference on Software Paradigm Trends* (ICSOFT-PT-2015), pages 121-129
ISBN: 978-989-758-115-1

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. There are many works on this topic. Some applies backward error recovery techniques (BER) that provide *Atomicity*, that is the case of Reconstructors (Fernández Lanvin et al., 2010). With that property in place, programmers can assume that modifications are done in an *all-or-nothing* way.

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

This paper describes the implementation of a tool that (1) translates invariants code (OCL) into executable code (AspectJ), (2) optimizes the constraints by generating simpler (incremental) versions regarding the events that affect the constraint, (3) delays the execution of those constraints until the close of the atomic context or a high-level operation, (4) is easy to integrate with atomic contexts such as Reconstructors, JPA transactions (Bauer et al., 2014) or STM (Harris et al., 2005) and (5) generates non-invasive code (aspect code).

The remaining of this paper is organized as follows: the second chapter summarizes the proposal, the third presents a running example, the tool is deep detailed in chapter four, chapter five shows some results, chapter six comments related work and chapter seven presents the conclusions.

2 PROPOSAL

We think that all the aforementioned difficulties could be avoided and automatized by means of appropriate consistency checking mechanisms that complement atomicity contexts. Checking all pending constraints when the atomic context is about to close solves the *when* difficulty (2). *What-to-do* (4) in case of failure is then solved due to the atomicity property of the context. The other two, *how* (1) and over *what-object* (3), can be solved applying OCL analysis techniques and code generation. Those techniques are able to convert the original constraints into new incremental versions (*how*) optimized for the events and objects affected (*what object*). Consequently, programmers' effort would be reduced.

Developers must implement the domain model classes as Plain Old Java Objects (POJO), and provide all constraints expressed in OCL in a separate unit.

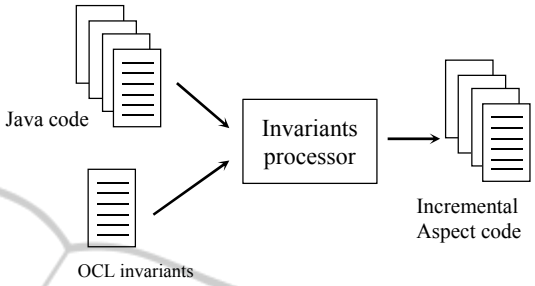

Figure 1: A tool to generate invariants code.

The tool (Figure 1) analyzes the model implementation and the OCL constraints. The constraint engine generates the code implementing an incremental checking of the given constraints. The output is the AspectJ code to be weaved with the programmer's code.

The programmer must also delimit the context of the business operations, in the same way he/she delimits transactions. At the end of the context every check will be done. If any constraint is violated, an exception will be raised indicating a constraint violation in the business operation.

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

Listing 1: Execution of business code inside a context.

Ideally this consistence integrity checking will be done in combination with some kind of atomicity handling context able to restore the model to its previous state. The tool currently integrates with Reconstructors (Fernández Lanvin et al., 2010), and with the Hibernate ORM (Bauer et al., 2014). The design would easily integrate also software transactional memory solutions (Harris et al., 2008; Dice et al., 2006).

3 RUNNING EXAMPLE

In order to illustrate different stages of the

constraints processing we will use a running example based on the well-known Royal&Loyal model proposed by Jos Warmer and Anneke Kleppe (Warmer and Kleppe, 2003). We show a reduced version of the Royal&Loyal system in Figure 2.

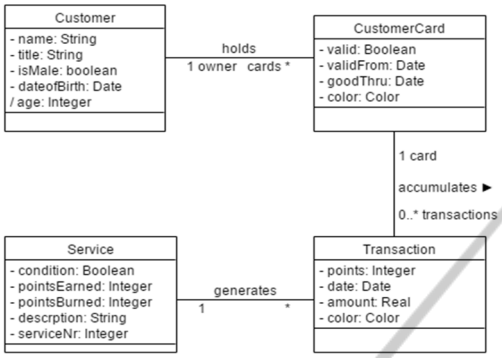

Figure 2: Reduced version of Royal&Loyal model.

Consider the following example restrictions expressed as OCL invariants over this domain model:

```
-- I1 Owner's card must be an adult
context CustomerCard inv I1:
 validFrom.diffYears(owner.dateOfBirth)
   >= 18
-- I2 Every senior citizen's card must
have a positive credit in all his
transactions
context Customer inv I2:
    self.age() >= 65
implies
   self.cards
    ->forAll(c | c.transactions
       ->collect(t | t.points )
       ->sum() >= 0)
```

4 THE TOOL

The tool makes use of Dresden OCL ToolKit (DOT) (Claas Wilke and Michael Thiele, 2010). 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.

Over the AST representation of the constraint we apply the algorithms proposed by (Cabot and Teniente 2009), a transformation method of OCL constraints into incremental and simpler versions. That is, if the system is currently in a valid state and we apply some modification over it, we do not need to check all constraints over all instances (that would be extremely inefficient), but just over the instances affected and only those constraints that could possibly be violated. Their algorithm transforms the original constraints into an equivalent set of simpler and optimized constraints according to the type of modification (event type).

4.1 Processing Every Constraint

The processing, applied over every constraint AST, consists of several stages as depicted in Figure 3.

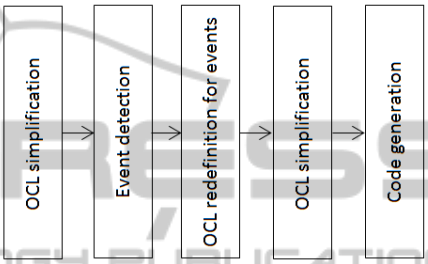

Figure 3: Constraint processing.

4.1.1 Constraint Transformation

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 relationship of this equivalences can be found in (Cabot and Teniente, 2007). 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. For this we follow the process explained in (Cabot and Teniente, 2009). 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 (Cabot and Teniente 2009).

As a result of this process we end up with some simpler constraints regarding every event for each constraint. These new constraints will be simpler and consequently, more efficient in execution.

Consider again the invariant example I1, it is affected by the events *UpdAtt(CustomerCard.validFrom), Link(Holds), Insert(CustomerCard)* and *UpdAtt(Customer.dateOfBirth)*. Being the two last events better checked by the redefinition I1-2 of the original invariant.

```
context Customer inv I1-2:
   self.cards->forAll(v |
     v.validFrom.diffYears(
        v.owner.dateOfBirth )>= 18)
```

The invariant I2 is affected by the events *Link(Accumulates), Unlink(Accumulates), Link(Holds)* and *UpdAtt(Transaction.points)*. For all those events the I2-2 redefinition is better focused then the original.

```
context CustomerCard inv I2-2:
   not (self.customer.age() >= 65
   or self.transactions->
     collect(t | t.points)->sum() >= 0
```

4.2 Code Generation for Event Detection

The output of the previous processing is a set of classes, with the events and the invariants that should be checked. The tool generates aspect code to detect those events over the objects that form the domain graph.

4.2.1 Attribute Modification

To detect this type of modification we create pointcuts following this pattern:

```
protected pointcut <att>Set(<cl> obj):
   set(* <cl>.<att>) && this(obj);
```

Where `<cl>` and `<att>` are placeholders for the class name and the attribute name. We advise that pointcut with a block of code as shown in Listing 2.

```
after(<cl> obj): <att>Set(obj) {
if  (!ContextFactory.hasActiveContext())
    return;
Method method = getInvariantMethod(
   <cl>.class, "<invariant_name>");
ContextFactory
   .getCurrentContext()
   .add( new Invariant(obj, method) );}
```

Listing 2: Insertion into the context of an invariant checker method after an event detection.

That code creates and registers an object in charge of checking and invariant when invoked (*new Invariant(...)*). This object will receive as argument the affected object and the reflective representation of the checking method to be executed. It is then stacked on the context waiting for context *close()* operation to be executed.

4.2.2 Linkage of Objects

We need to distinguish between linking to unique association ends and many association ends. The former are represented in Java by a reference to an object, while the latter are usually supported by a collection. One-side linking is detected with the same pattern as for attribute modification detection.

In case of a many side, we need to detect additions and removals to the underlying collection. The tool introduces a proxy for the collection to generate callbacks when a modification to the underlying collection is done. That proxy must be configured with the event types it has to notify (additions or deletions). It is injected after the assignment of the collection to the object's field during construction time. The pointcut we use here follows a pattern like this:

```
pointcut <att>ColSetter(<cl> obj):
   set(Set <att>)
   && within(<cl>)
   && target( obj );
```

With and *after* advise for that pointcut the proxy is inserted:

```
after(<cl> obj) : <cl>ColSetter(obj){
Field field =
   getField(<cl>.class, "<att>");
IncContainer ic = newProxy(obj, field);
applyValueToField(obj, field, ic);

ContainerEvent[] events;
Method method;
method = reflectivelyGetMethod(
   <cl>.class, "<invariant>");
events = new ContainerEvent[]{<evnts>};
InvariantBuilder builder =
   new InvariantBuilder(obj, method);
ic.registerInvariantBuilderForEvents(
      builder, events);
}
```

Listing 3: After advise pattern for many side linking and unlinking

The last line of code configures the proxy with the events it must notify and an invariant builder object whose mission is to create and insert into the context an invariant checker (new Invariant(…)) whenever one of the specified events occurs.

4.3 Creation and Deletion of Objects. Extent of a Class

Those OCL constraints that involve an *allInstances* expression are especially difficult to compute due to the fact that not all created objects are valid objects. Just by detecting the construction of an object (with a pointcut 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. Several questions comes 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 that are reachable from the graph. In that way we can detect the addition of a new object when it is linked to another object already in the graph. Conversely an object deletion will be produced after the removal of all links that maintain the object linked to the graph.

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.

```
privileged aspect GraphNodeAspect {
   boolean GraphNode.isInRepository;
   List<GraphNode> GraphNode.forward;
   List<GraphNode> GraphNode.backward;
   …
}
```

When an insertion or deletion (Insert or Delete events) is detected the affected object is then added or removed to/from the corresponding *allInstances* collection.

```
void GraphNode.removeFromAllInstances(){
  Extents.get( this.getClass() )
     .remove(this);
  for(GraphNode entity: forward) {
    if ( ! entity.isInGraph() ) {
      entity.removeFromAllInstances();
    }}}
```

We already know how to detect when two objects are linked or unlinked. Now we need to augment the body of the previous after advise pattern (Listing 3) to check the reachability of both objects after the link/unlink operation.

```
after(<cl> obj) : <cl>ColSetter(obj){
 // same code as Listing 3

 // extra code to detect Insert
 events = new ContainerEvent[]{Insert};
 method = getInvariantCheckerMethod(
    <cl>.class, "<invariant>");
}
```

The system maintains a collection for every domain class. The contents of these collections are updated after the additions and removals.

There is one remaining question. A graph is a set of interrelated objects, but there could be many independent graphs. What is the real graph? In our conception the graph must have some root nodes (objects) to which other objects are connected after. Those root nodes are usually well localized in the design and stored in some type of collection (Repository pattern in (Evans, 2003)). With that in place we can state the condition an object must meet to be considered in-the-graph: *An object will be in the graph if it is directly stored in a repository or is reachable from another object that is already in the graph.*

```
public boolean GraphNode.isInGraph() {
  return isInRepository
         || anyRelatedIsInGraph();}
boolean GraphNode.anyRelatedIsInGraph()
{
  for(GraphNode n: backward) {
    if (n.isInGraph()
      && n.forward.contains( this )) {
      return true;}
  }
  return false;}
```

In this tool we use an annotation (@Repository) to mark those collections that act as repositories.

Finally, by proxying those collections with and aspect advise we are able to detect additions or removals of root objects. Whenever a new object is added its *inRepository* attribute is set to true, and the opposite when it is removed. Consequently, all objects reachable from this object will acquire or lose their *in-the-graph* condition by reachability (and will fire the respective Insert/Delete event).

4.4 Code Generation of Invariants

Once the invariants have been transformed in their

incremental versions they can be translated into Java. For this step we use again the Dresden OCL Toolkit. The output DOT code generator is a list of strings, being the last one the final Boolean expression used to raise and exception in case it evaluates to false. The invariant checker method will follow this pattern:

```
public void <inv_name>(<class> obj) {
  <DOT generated lines>
  if (! <DOT generated last line>) {
    throw new ConstraintException(…);}}
```

4.5 Execution Context

All business operations must be executed within a context (similar idea as a transaction). This functionality is represented by the Context interface.

```
public interface Context {
    public void add(Invariant i);
    public void close() throws …
    public void dispose() throws …
}
```

The developer must invoke the business operations within an opened context as shown in Listing 1. Explicitly context handling can be avoided by annotating the business methods with @Context. The tool put the context handling code behind the scenes.

```
@Context public void doBusiness(…){…}
```

Context objects are obtained from a context factory class that maintains the context object linked to the current running thread. The *add()* method is invoked from the event detectors to insert the corresponding invariant checker method that, along with other pending invariants, must be checked at the end of the context (when the *close()* method is called).

4.5.1 Event Simplification

During the execution of the business operation some events may arise, and consequently the event handler inserts an invariant checker method to verify the corresponding constraint. The context stores every checker object classified by its origin object, event type and invariant method to call. With this information in place there are some optimizations that can be done to improve efficiency.

- In the case of UpdAtt events repeated over the same object attribute, the context just store one. If there is a previous Insert event then the UpdAtt is irrelevant, as all invariant checkings related to UpdAtt events are always verified by an Insert event.
- With Delete events we must delete all previous invariants for the same instance. Besides, if there already an Insert event for the same entity we do not even need to store the Delete event.
- The case of Unlink event is similar to the previous one, if there already is a Link event for the same association and object in the context the Link event must be deleted. And if Link and Unlink are in the same context, none of them deserve to be checked.
- Finally, as different events could raise the same invariant checking, the context object should avoid registering the same object-invariant more than once.

4.5.2 Final Execution

When the business operation is finished, the context is closed and every pending check is executed. The context catches and stucks every possible violation. After that, all the accumulated exceptions are gathered together in one final exception raised with all that information in place. That way the programmer can obtain information about every broken constraint in one single shot.

5 RESULTS

We have tested our tool with the full version of the Royal&Loyal model already presented (Warmer & Kleppe, 2003). For that purpose we take the invariant definitions available as example in the Dresden Toolkit (Claas Wilke et al., 2009). The full domain model consists of 11 entity classes and 2 extra types. It also has 20 OCL invariants of which 6 are of attribute or object type, 12 are of domain type and 2 of class type.

Consider a use case in which a *Customer* consumes a *Service* offered by a *Program Partner* of a *Loyalty Program* to which both are associated and is paid with the points accumulated on the *Loyalty Account* by the previous customer's *Transaction*s. During the operation the system has to register a new *Burning* transaction for a number of points specified by the service. Listing 4 shows the involved invariants.

```
(1) context Burning
inv burningAsTransaction: points =
      oclAsType(Transaction).points
(2) context ProgramPartner
inv totalPointsEarning:
```

```
  self.services.transactions
    ->select(t| t.oclIsTypeOf(Earning))
    ->collect( tt | tt.points)
    ->sum() < 10000
(3) context ProgramPartner
inv totalPoints:
   self.services.transactions
      ->collect(t | t.points)
    ->sum() < 10000
(4) context LoyaltyAccount
inv oneOwner:
 self.transactions.card.owner->size()=1
(5) context LoyaltyAccount
inv transactionsWithPoints:
   self.points <= 0
   or self.transactions
      ->select(t | t.points > 0)
    ->size() > 0
```

Listing 4: Invariants generated.

As discussed before, in case of using a DbC approach the object's invariants would only be checked due to object's methods executions, and thus the invariant's affected object could be unware of possible invariants violations due to changes in other linked objects. As shown in Table 3, just one invariant is of attribute or object type, therefore the other invariants will not be checked (unless some other methods of the related *ProgramPartner* and *LoyaltyAccount* objects are executed).

Alternatively we can use an OCL interpreter, widely used in some scenarios such as model to model transformations. An interpreter checks all the constraints against all objects in the model instance. We can use the amount of objects visited as an indicator for comparing the three approaches mentioned.

After executing the tool we get 36 new invariants related to 25 affecting events and 11 new AspectJ files ready to be weaved with the entities[1].

During the execution of the use case, several affecting events will be produced indicating a potential violation of their related constraints.

Table 1: Events raised by the use case execution.

Ev id	Ev type	Over entity type
1	Insert	Transaction (base class of Burning)
2	Insert	Burning
3	Link	Transaction and Service
4	Link	Transaction and CustomerCard
5	Link	Transaction and LoyaltyAccount
6	UpdAtt	LoyaltyAccount.points

[1] The tool and the all related code for this testing can be downloaded from http://www.di.uniovi.es/~alberto_mfa/constraints.proto.zip

Table 2: Invariants stacked onto the context due to the previous events (Id column relates with the id column of Table 1).

Ev Id	Context	Invariant
2	Burning	burningAsTransaction
3	ProgramPartner	totalPointsEarning
3	ProgramPartner	totalPoints
4	LoyaltyAccount	**oneOwner**
5	LoyaltyAccount	**oneOwner**
6	LoyaltyAccount	transactionsWithPoints-19

In the Table 2 we can observe that event 1 has no invariant associated, while event 3 has two of them. Besides, the oneOwner-24 invariant is raised by two different events. Thanks to context optimization those repetitions are avoided and eventually only 5 invariants require to be checked.

Table 3: Type of each invariant and number of objects accessed by each one (id refers to Listing 4). The symbol (.) indicates the formula in the "Proposed Tool" column.

Inv Id	Inv Type	Proposed Tool	OCL intrpr.
1	Attribute	1	N_B * (.)
2	Domain	$1 + S_{PP} * (1 + T_S)$	N_{PP} * (.)
3	Domain	$1 + S_{PP} * (1 + T_S)$	N_{PP} * (.)
4	Domain	$1 + (3 * T_{LA})$	N_{LA} * (.)
5	Domain	$1 \parallel 1 + T_{LA}$	N_{LA} * (.)

Table 3 relates the invariant, the type and number of objects accessed for its verification. The third column indicates the number of objects using the proposed tool while the fourth do the same for an OCL interpreter. Here S_{PP} stands for the average number of *Service* objects linked to a *ProgramPartner* object, T_S represents the average number of *Transactions* linked to a *Service*, and T_{LA} means the average number of *Transactions* linked to a *LoyaltyAccount*.

The OCL interpreter must execute each invariant for every context class object in the system. That is represented in the right column where N_B stands for the total number of Burning transactions in the system; N_{PP} represents the total number of *ProgramPartners* and NLA the total of *LoyaltyAccounts*.

6 RELATED WORK

This idea of objects having to satisfy a set of invariants traces back to the work of Hoare (Hoare, 1972). Later Meyer continued the idea with his *Design by Contract* (DbC) methodology (Meyer, 1992). Nowadays this idea was also applied to many other languages such as JML for Java (Leavens &

Cheon, 2005), Spec# for C# (Barnett et al., 2004), etc.

Design by Contract is based on the principle of an object being responsible for its own consistency. This rule is practical for single objects not associated with others (attribute and object constraints in our classification), or just having references to its owned objects (composition), but does not match with class and domain constraints. Therefore, DbC is enough for attribute and object constraints, but is not practical for class and domain constraints.

There are also many works using OCL based contracts. Some tools translate them into Java, AspectJ (Cheon et al., 2008; Gopinathan & Rajamani, 2008; Dzidek et al., 2006; Rebêlo et al., 2008) or other contract languages such as JML (Avila et al., 2008; Hamie, 2004) or CleanJava (Cheon & Avila, 2010). All this works differ from our approach in their adherence to DbC (attribute and object constraints only). However, those that generate AspectJ suggest techniques and templates. In (Froihofer et al., 2007) the authors offers a complete report and comparison of those techniques. We take the idea of using proxies for them.

Henrique Rebêlo et al. (Leavens et al., 2014), propose a JML to AspectJ compiler able to solve one the problems addressed with our proposal, the scattering of the contract specification among different methods that may violate it. Their work avoids contract scattering by centralizing the contract specification in a common advice complemented with JML. Our approach also avoids scattering and promotes the invariants specification as documentation by centralizing all invariants in one single file.

Dzidel et al. (Dzidek et al., 2006) present another OCL-contract to AspectJ tool, but leave as future work some problems we try to solve with our proposal: (1) the challenge of translating the OCL *allInstances* expression into target code, and (2) the runtime overhead of checking OCL collection expressions as *forAll*, *collect*, etc. We have proposed a possible solution to the *allInstances* problem using the idea on being *in-the-graph*.

Another type of OCL tools are the interpreters (Chimiak-Opoka et al., 2011). They are aimed to check a model instance against its model and constraints. That may seem a solution but they work in a one shot fashion: they check every constraint against the whole model instance. This solution is practical for those situations in which the whole model instance is created at once, for example in model transformations (MDA). But this strategy will lead to unfeasible performance rates for a domain model being incrementally updated by business logic method executions.

A common point in all these DbC and OCL tools is that they do not perform any analysis of constraints, thus the generated code is not incremental. Although some of them can generate code able to detect plain attribute modifications, they insert the checking right after the modification (Claas Wilke et al., 2009) or allow the programmer to call the checking method later, leading to the programmer the responsibility to explicitly decide *when* and *what* method to call. They help with the *how* difficulty, partially with the *over-what* object, but neither with the *when* nor with the *what-to-do* difficulties.

7 CONCLUSIONS

The proposed tool aids developers with the four discussed difficulties. The generated code is able to detect those potentially affecting events (*what object*) which combined with the delayed checking (*when*) and the transformed invariants translated to executable code (*how*) is a key difference with all DbC-like implementations for the specific case of programs built around the domain model pattern. The integration with atomicity contexts such as Reconstructors (Fernández Lanvin et al., 2010) or Hibernate (Bauer et al., 2014) solves the problem of restoring the model to a previous state (*what-to-do*), although that integration is not mandatory; the generated code could work without that capability.

As we can conclude from results section the efficiency is quite good. Due to the incremental approach followed, every constraint is executed over as few objects as necessary and the context simplification process may reduce the number of constraint checkings.

The tool also gives a possible implementation for the *allInstances* problem, a classical problem when translating OCL to Java code.

Finally, by maintaining all invariants in a single source file it also helps with the problem of invariant scattering while it preserves the invariants as documentation for programmers.

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).

REFERENCES

Avila, C., Flores, G. & Cheon, Y., 2008. A library-based approach to translating OCL constraints to JML assertions for runtime checking. In *International Conference on Software Engineering Research and Practice, July 14-17, 2008, Las Vegas, Nevada*. Citeseer, pp. 403–408.

Barnett, M., Leino, K.R.M. & Schulte, W., 2004. The Spec\# Programming System: An Overview. In *International Conference in Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04)*. Springer, pp. 49–69.

Bauer, C., King, G. & Gregory, G., 2014. *Java Persistence with Hibernate* Manning Publications Co., ed., Manning Publications Co.

Cabot, J. & Teniente, E., 2009. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, 82(9), pp.1459–1478.

Cabot, J. & Teniente, E., 2007. Transformation techniques for OCL constraints. *Science of Computer Programming*, 68(3), pp.179–195.

Cachopo, J.M.P., 2007. *Development of Rich Domain Models with Atomic Actions*. UNIVERSIDADE TÉCNICA DE LISBOA.

Cheon, Y. et al., 2008. An aspect-based approach to checking design constraints at run-time. In pp. 223–228.

Cheon, Y. & Avila, C., 2010. Automating Java program testing using OCL and AspectJ. In *ITNG2010 - 7th International Conference on Information Technology: New Generations*. pp. 1020–1025.

Chimiak-Opoka, J. et al., 2011. OCL Tools Report based on the IDE4OCL Feature Model. *Eceasst*, 44.

Claas Wilke, Dr.-Ing. Birgit Demuth & Prof. Dr. rer. nat. habil. Uwe Aÿmann, 2009. Java Code Generation for Dresden OCL2 for Eclipse.

Claas wilke & michael thiele, 2010. DRESDEN ocl2 for eclipse Manual for Installation, use and Development.

Dice, D., Shalev, O. & Shavit, N., 2006. Transactional Locking II. *Distributed Computing*, 4167, pp.194–208.

Dzidek, W.J., Briand, L.C. & Labiche, Y., 2006. Lessons learned from developing a dynamic OCL constraint enforcement tool for java. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3844 LNCS, pp.10–19.

Evans, E., 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional.

Fernández Lanvin, D. et al., 2010. Extending object-oriented languages with backward error recovery integrated support. *Computer Languages, Systems & Structures*, 36(2), pp.123–141.

Fowler, M., 2003. *Patterns of Enterprise Application Architecture* I. Addison-Wesley Longman Publishing Co., ed., Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Froihofer, L. et al., 2007. Overview and evaluation of constraint validation approaches in Java. In *Proceedings - International Conference on Software Engineering*. IEEE, pp. 313–322.

Gopinathan, M. & Rajamani, S.K., 2008. Runtime monitoring of object invariants with guarantee. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5289 LNCS, pp.158–172.

Hamie, A., 2004. Translating the Object Constraint Language into the Java Modelling Language. *Proceedings of the 2004 ACM symposium on Applied computing - SAC '04*, p.1531.

Harris, T. et al., 2005. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '05*. New York, New York, USA: ACM Press, p. 48.

Harris, T. et al., 2008. Composable memory transactions. *Communications of the ACM*, 51(Section 2), p.91.

Hoare, C.A.R., 1972. Proof of correctness of data representations. *Acta Informatica*, 1(4), pp.271–281.

Leavens, G.T. et al., 2014. AspectJML : Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proceedings of the 13th International Conference on Modularity*. Lugano, Switzerland: ACM, pp. 157–168.

Leavens, G.T. & Cheon, Y., 2005. Design by Contract with JML. *Draft, available from jmlspecs. org*, 1, p.4.

Meyer, B., 1992. Applying `design by contract'. *Computer*, 25(10), pp.40–51.

Meyer, B., 2015. Framing The Frame. *NATO Science for Peace and Security*, (Series D: Information and Communication Security), pp.174–185.

Olivé, A., 2005. Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In O. Pastor & J. Falcão e Cunha, eds. *Advanced Information Systems Engineering*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1–15.

Rebêlo, H. et al., 2008. Implementing Java Modeling Language contracts with AspectJ. *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pp.228–233.

Warmer, J. & Kleppe, A., 2003. *The Object Constraint Language: Getting Your Models Ready for MDA* 2nd ed., Addison-Wesley Professional.