Invariant Implementation for Domain Models

Applying Incremental OCL Techniques

Alberto-Manuel Fernández-Álvarez, Daniel Fernández-Lanvin, and Manuel Quintela-Pumares

Computing Science Department, University of Oviedo, Oviedo, Asturias, Spain
{alb, dflanvin, quintelamanuel}@uniovi.es

Abstract. Constraints for rich domain models are easily specified with the Object Constraint Language (OCL) at model level, but hard to translate into executable code. We propose a solution which automatically translates the OCL invariants into aspect code able to check them incrementally after the execution of a Unit of Work getting good performance, a clean integration with programmers' code being respectful with the original design and easily combined with atomic all-or-nothing contexts (data base transactions, STM, ORM, etc.). The generated code solves some difficult issues: how to implement, when to check, over what objects and what to do in case of a violation.

1   Introduction

Domain modeling is a well-known practice to capture the essential sematic of a rich domain. The advantages of this approach have been widely discussed in the software engineering literature [1–3]. 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. However, those that affect to more than one object of the domain (domain constraints) are determined by the state and relationships of every con-

cerned 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. There may be several methods that need to check the constraint, or part of it, as a postcondition. That could lead to code scattering, code tangling and thigh coupling [4].

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*) that 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 over what objects they should be checked and then, at the end of the high-level method call 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* [5]. 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 can suppose a source of implementation problems in the project.

This paper describes the implementation of a tool that (1) translates invariants code (OCL) into executable code (aspect code), (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* or an STM and (5) generates non-invasive code (aspect code), and thus easy to add to programmer's code.

The remaining of this paper is organized as follows: the second chapter summarizes the proposal, the third presents the difficulties a programmer must solve, the fourth introduces a running example the tool is deep detailed in chapter five and chapter six shows the results of applying the proposed processing to the running example, the seventh chapter comments related work and the eighth closes with 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.
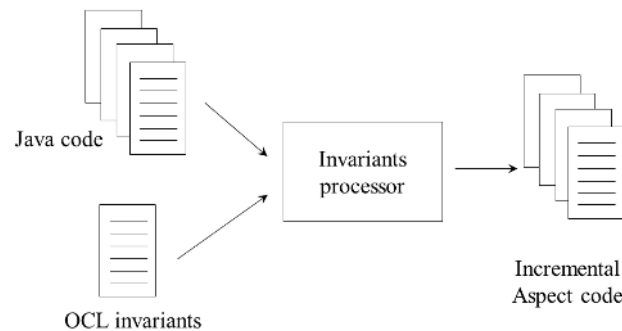
Fig. 1. A tool to generate invariants checking code.

The tool (Fig. 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();
}
```

Fig. 2. 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* [5], and with the Hibernate ORM [6]. The design would easily integrate also software transactional memory solutions [7, 8].

3 Constraint Implementation Issues

Depending on the model elements involved we could classify constraints on four different categories: (1) attribute, (2) object, (3) class and (4) domain constraints.

Let us use the example in Fig. 3 for illustrating purposes in which we add three constraints as OCL invariants to a small UML model.
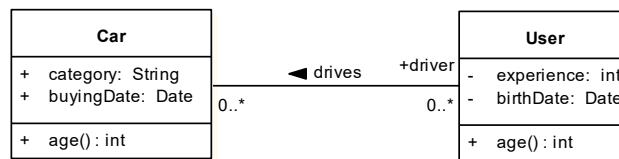

Fig. 3. Car and driver UML diagram.

```
-- I1: The age of a user must be a positive value
context User inv I1:
    self.age() > = 0


-- I2: If a car is gold class then it must be
-- less than 4 years old
context Car inv I2:
    self.category = 'gold' implies self.age() < 4


-- I3: There can only be 10 cars of gold class at maximum
context Car inv I3:
    Car.allInstances->select(category ='gold')->size() <10


-- I4: Gold class cars can only be driven by
-- drivers with more than 10 years of experience.
context Car inv I4:
  self.category = 'gold' implies self.drivers->forAll(d |
        d.experience > 10
      )
```

Attribute constraints are those that only depend on the value of an object attribute, for example the I1 constraint. *Object constraints* involve several attributes of the same object, as example the I2. Here the constraint depends on the category and age attributes. *Class constraints* (I3) depend on all instances of a class (extent). *Domain constraints* are those affected by the state of a subgraph of objects (I4).

Attribute and object constraints can be safely checked after any attribute modification. Those changes are typically done by object methods and, thus, the enumerated difficulties have a straight solution. These constraints can be checked after (*when*) the execution of the object method that changes the attribute (as a postcondition). The implementation of the constraint checking will be a plain translation into the imple-

mentation language (*how*). The concerned object is clearly located (*what object*). The last issue (*what to do*) may also be easily solved provided we have an atomic execution context.

Class constraints are more difficult to check properly. As more than the currently-being-modified object is involved, the constraint is not only sensitive to the modification of object's attributes it depends on, but also to insertions and deletions of other objects. When to check the constraint has more affecting events (updates, inserts and deletes) and the *what-object* difficulty is also more complex as more objects are involved. The complexity of the implementation (*how*) of this kind of constraints is related to the architecture; in the case of plain model to code translation, it is usually hard to maintain the extent of the class. Not many languages maintain such a collection of objects, and in those with garbage collection, it is hard to control if an object has been functionally deleted (unreferenced) or not.

Domain constraints management involves even more problems. Several linked objects may be involved, so any change in any of them might violate the constraint. Let's see a trivial example of two entities, A and B, which have a bidirectional association. There is an implicit constraint: if an object of A has a reference to an object of B, then this object of B must have a reference to the A object. If we check the constraint after setting the reference to B in A (a.setB(b)), the object A will be in a valid state, but B is not linked to A yet, thus the constraint is not meet. If we do the reverse we end in a similar situation. The obvious solution here is to delay the checking until the high-level operation has finished (*when*). As the constraint might be affected by different objects, through different method calls, there could be various objects on which we have to check the constraint (more difficult over *what-object*). The implementation of the constraint (*how*) have to consider more than one class. Combining this with every possible method that may affect it would lead to a combinational explosion. That kind of situations are error-prone since a programmer can easily forget some of those methods. This quickly leads to code scattering, code tangling, and strong coupling of code [4]. Besides, as an attribute may be involved in several constraints (in the previous examples the car class attribute), the implementations of the modifier methods will end up even more tangled.

All those difficulties acquire a new dimension if performance is another concern. The programmer should analyze the constraint looking for affecting modifications (relevant events). Consider again the example constraint I4 "gold class cars can only be driven by drivers with more than 10 years of experience". We will conclude that we only need to check that constraint if there is an updating of the car class attribute, or a change in the driver experience field, or when a car and a driver are linked. The rest of all possible modifications in the model will never affect it, so it is unnecessary to check the constraint. Note that checking the constraint after an effective attribute modification is more precise than checking it after a method execution that might have changed the attribute or not.

Another efficiency gap comes from the checking of more objects than those really affected by the changes. For instance, to check the constraint over all cars after modifying the category attribute in one of them clearly has no point. Note that the processing of the relevant instances for a constraint must consider not only the entities of

the context type directly modified by the structural events, but also those entities related directly or indirectly to certain modified entities of other types.

Furthermore, depending on the type of event, the body of the constraint could be different, better focused. For instance, if the case of a linking event between a car and a driver, then an equivalent and more efficient constraint body would be:

```
-- I4: better focused for class-driver linking
context Drives inv:
    self.car.category <>'gold' or self.driver.experience>10
```

That redefinition only requires the two linked objects. Neither needs to check all the drivers related with the car nor all the cars related with the driver.

As another example for the case of User::experience modification, we could check this other redefined constraint:

```
-- I4: better focused for driver.experience changes
context User inv: self.cars->forAll(c |
        c.category <> 'gold' or self.experience > 10
    )
```

This performs better given that it only involves the cars related with the driver. Note the context change from Car to User.

As a conclusion, we can state that implementing constraints avoiding code tangling, scattering or high coupling, with an efficient (incremental) checking and ensuring the stability of the system in case of constraint violation is not a trivial task for the developer.

4   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 [9]. We show a reduced version of the *Royal&Loyal* system in Fig. 4.
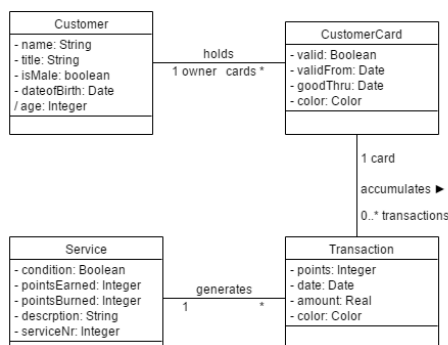

Fig. 4. Reduced version of the Royal&Loyal model.

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
   )
```

5   The Tool

The tool makes use of Dresden OCL ToolKit (DOT) [10]. DOT provides a set of tools to parse and evaluate OCL constraints on various models like UML, EMF and Java. The tool 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 that AST representation we apply the processing proposed by [11]. They propose an incremental checking of constraints. I.e., 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 by the modification. The algorithm transforms the original constraints into an equivalent set of simpler and optimized constraints according to the type of modification (event type).

5.1   Processing Every Constraint

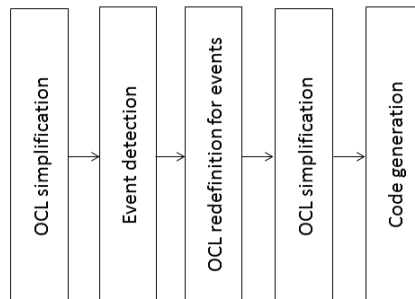The processing of constraint AST consists of several stages, as depicted in Fig. 5.

Fig. 5. 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 relationship of this equivalences can be found in [12]. 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 [11]. 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 [11].

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
```

5.2 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.

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 Fig. 6.

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

Fig. 6. Insertion into the context of an invariant checker method after 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.

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) : <att>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);
}
```

Fig. 7. *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.

5.3 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 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 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 (Fig. 7) to check the reachability of both objects after the link/unlink operation.

```
after(<cl> obj) : <cl>ColSetter(obj){
    // same code as Fig. 7
    // extra code to detect Insert
    events = new ContainerEvent[]{Insert};
    method = getInvariantMethod(<cl>.class, "<inv_name>");
}
```

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 [1]). 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. Consequent-

ly, 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).

5.4   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(...);}}
```

5.5   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 Fig. 2. 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 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).

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.

Final Execution.
When the business operation is finished, the context is closed and every pending check is executed. The context catches and stacks 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.

6   Results

We have tested our tool with the full version of the *Royal&Loyal* model already presented [9]. For that purpose we take the invariant definitions available as example in the Dresden Toolkit [13]. 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. Fig. 8 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
```

Fig. 8. New generated versions of invariants.

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 unaware 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

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
1		
2	Burning	burningAsTransaction
3	ProgramPartner	totalPointsEarning
3	ProgramPartner	totalPoints
4	LoyaltyAccount	**oneOwner**
5	LoyaltyAccount	**oneOwner**
6	LoyaltyAccount	transactionsWithPoints-19

In Table 2 we can observe that event 1 has no invariant associated, while event 3 has two of them. Besides, the oneOwner invariant is raised by two different events.

---

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

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 Table 1). 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 SPP stands for the average number of Service objects linked to a ProgramPartner object, TS represents the average number of Transactions linked to a Service, and TLA 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 N_{LA} the total of *LoyaltyAccounts*.

7   Related Work

This idea of objects having to satisfy a set of invariants traces back to the work of Hoare [14]. Later Meyer continued the idea with his Design by Contract (DbC) methodology [15]. Nowadays this idea was also applied to many other languages such as JML for Java [16], Spec# for C# [17], 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 [18–21] or other contract languages such as JML [22, 23] or CleanJava [24]. 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 [25] the authors offers a complete report and comparison of those techniques. We take the idea of using proxies for them.

Henrique Rebêlo et al. [26], 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. [20] 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 [27]. 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 [13] 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.

8   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* [5] or Hibernate [6] 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 verifications.

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

References

1. Evans, E.: Domain-Driven Design-Tackling Complexity in the Heart of Software. Addison-Wesley Professional (2003).
2. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA (2003).
3. Olivé, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In: Pastor, O. and Falcão e Cunha, J. (eds.) Advanced Information Systems Engineering. pp. 1–15. Springer Berlin / Heidelberg (2005).
4. Cachopo, J.M.P.: Development of Rich Domain Models with Atomic Actions, UNIVERSIDADE TÉCNICA DE LISBOA (2007).
5. Fernández Lanvin, D., Izquierdo Castanedo, R., Juan Fuente, A.A., Fernández Álvarez, A.M.: Extending object-oriented languages with backward error recovery integrated support. Comput. Lang. Syst. Struct. 36, 123–141 (2010).
6. Bauer, C., King, G., Gregory, G.: Java Persistence with Hibernate. Manning Publications Co. (2014).
7. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '05. p. 48. ACM Press, New York, New York, USA (2005).
8. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. Distrib. Comput. 4167, 194–208 (2006).
9. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Professional (2003).
10. Claas Wilke, Michael Thiele: DRESDEN OCL2 FOR ECLIPSE MANUAL FOR INSTALLATION, USE AND DEVELOPMENT, (2010).
11. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software, 82(9), pp.1459-1478, (2009).
12. Cabot, J., Teniente, E.: Transformation techniques for OCL constraints. Sci. Comput. Program. 68, 179–195 (2007).
13. Claas Wilke, Dr.-Ing. Birgit Demuth, Prof. Dr. rer. nat. habil. Uwe Aÿmann: Java Code Generation for Dresden OCL2 for Eclipse, http://dresden-ocl.sourceforge.net/downloads/pdfs/gb_claas_wilke.pdf, (2009).
14. Hoare, C.A.R.: Proof of correctness of data representations. Acta Inform. 1, 271–281 (1972).
15. Meyer, B.: Applying "design by contract." Computer (Long. Beach. Calif). 25, 40–51 (1992).
16. Leavens, G.T., Cheon, Y.: Design by Contract with JML. Draft. available from jmlspecs. org. 1, 4 (2005).
17. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec\# Programming System: An Overview. In: International Conference in Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '04). pp. 49–69. Springer (2004).

18. Cheon, Y., Avila, C., Roach, S., Munoz, C., Estrada, N., Fierro, V., Romo, J.: An aspect-based approach to checking design constraints at run-time. Presented at the November (2008).

19. Gopinathan, M., Rajamani, S.K.: Runtime monitoring of object invariants with guarantee. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). 5289 LNCS, 158–172 (2008).

20. Dzidek, W.J., Briand, L.C., Labiche, Y.: Lessons learned from developing a dynamic OCL constraint enforcement tool for java. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). 3844 LNCS, 10–19 (2006).

21. Rebêlo, H., Soares, S., Lima, R., Ferreira, L., Cornélio, M.: Implementing Java Modeling Language contracts with AspectJ. SAC '08 Proc. 2008 ACM Symp. Appl. Comput. 228–233 (2008).

22. Avila, C., Flores, G., Cheon, Y.: 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. pp. 403–408. (2008).

23. Hamie, A.: Translating the Object Constraint Language into the Java Modelling Language. Proc. 2004 ACM Symp. Appl. Comput. - SAC '04. 1531 (2004).

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

25. Froihofer, L., Glos, G., Osrael, J., Goeschka, K.M.: Overview and evaluation of constraint validation approaches in Java. In: Proceedings - International Conference on Software Engineering. pp. 313–322. IEEE Computer Society, Los Alamitos, CA, USA (2007).

26. Rebêlo, H., Leavens, G.T., Bagherzadeh, M., Rajan, H., Lima, R., Zimmerman, D.M., Cornélio, M., Thüm, T.: AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In: Proceedings of the 13th International Conference on Modularity. pp. 157–168. ACM, Lugano, Switzerland (2014).

27. Chimiak-Opoka, J., Demuth, B., Awenius, A., Chiorean, D., Gabel, S., Hamann, L., Willink, E.D.: OCL Tools Report based on the IDE4OCL Feature Model. Eceasst. 44, (2011).