On Preserving Domain Consistency for an Evolving Application*

João Roxo Neves and João Cachopo

INESC-ID / Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal
{JoaoRoxoNeves,joao.cachopo}@ist.utl.pt

Abstract. Many of today’s existing information systems are domain-intensive enterprise applications that manage and store vast volumes of complex data. Typically, these systems keep their data organized as a graph of highly interrelated objects, which must satisfy complex domain consistency rules. The Fénix Framework allows the programmer to define consistency rules, by implementing consistency predicates that objects must satisfy. The framework uses these predicates to verify the consistency of the data, as it changes over time.

Our previous work that introduced the consistency predicates did not take into account the aspects of persistence and incremental development, which are common in enterprise applications. Because these applications evolve, it is crucial to verify the consistency of the data as the code changes over time.

In this paper, we describe a few extensions to the Fénix Framework. We discuss why persistence and incremental development make the task of verifying these rules so difficult, and we propose pragmatical solutions to address these issues.

With this proposal, the system becomes aware of changes within consistency rules: it detects new and old rules, and takes action accordingly. Also, it supports inconsistency tolerance to help in establishing new consistency rules on already existing, and possibly inconsistent data.

Key words: Data Consistency, Consistency Rules, Invariants, Data Persistence, Inconsistency Tolerance, Change Impact Analysis

1 Introduction

Many modern enterprise applications must implement a rich, complex domain model, typically as a dense graph of classes and relationships in some object-oriented programming language [7,5]. Developing such a complex domain model is often a challenging task for most development teams.

To tackle this problem, we have previously proposed to use a Domain Modeling Language (DML) and a Software Transactional Memory (STM) to simplify the development of rich domain models [2,3].

* This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the project Cloud-TM (co-financed by the European Commission through the contract no. 257784).
One of the distinctive aspects of this approach is that it separates the implementation of a domain model in three parts: (1) its structure, implemented in the DML; (2) its behavior, implemented in Java; and (3) its consistency rules, implemented with *consistency predicates*.

Consistency predicates allow programmers to implement the consistency rules that the application’s data must satisfy independently of the implementation of the application’s behavior. They are checked automatically at the end of each business transaction to ensure that no consistency rule has been violated, in which case the transaction can commit; otherwise, the transaction aborts to prevent introducing inconsistencies in the application’s data.

As originally proposed, however, this approach assumed an execution model where the application starts from an empty state, keeps all of its data in transient memory, and runs forever without updates to its code. Clearly, this is not the case for enterprise applications, which keep persistent data and are incrementally developed over a long period of time. New consistency rules are often added for already existing data, which may be inconsistent according to the new rules.

In this paper, we extend the consistency predicates to take into account the pragmatic needs of a development team doing incremental development of an enterprise application in Java, where data is persisted and evolves with the application’s code. Our proposal was implemented as an extension to the Fénix Framework,¹ which implements our original approach and is used by several large-scale web applications [6].

More specifically, we describe how to keep the information about the existing consistency predicates across different executions of an application, how to detect changes in the implementation of consistency predicates, and how to deal with these changes. Moreover, we introduce the concept of *inconsistency tolerance* to handle existing inconsistent data that a new consistency rule reveals, and describe how the consistency predicates may be extended to support it.

The rest of the paper is structured as follows. In Section 2, we give an overview of how consistency predicates are implemented in the Fénix Framework and discuss some of their problems. Then, in Section 3, we describe how we extended the Fénix Framework to support changes in the consistency predicates of an application, and in Section 4, we present the new inconsistency tolerance semantics. In Section 5, we present related work. Finally, in Section 6, we draw some conclusions and discuss some future work.

2 Consistency Predicates in the Fénix Framework

Consistency predicates are defined by the programmer as plain Java methods within the classes that implement the application’s domain entities. These methods must return a boolean value, receive no arguments, and contain the @ConsistencyPredicate annotation. In Figure 1, we show an example of a consistency predicate for a hypothetical Client class.

¹ http://fenix-ashes.ist.utl.pt/trac/fenix-framework
public class Client {
    // ... some other banking-related code ...
    @ConsistencyPredicate
    public boolean checkTotalBalancePositive() {
        int totalBalance = 0;
        for (Account account : getAccounts()) {
            totalBalance += account.getBalance();
        }
        return (totalBalance >= 0);
    }
}

Fig. 1: A consistency predicate that checks that the Client’s total balance is positive. It calculates the sum of the balances of all of the Client’s Accounts.

A consistency predicate invoked on a domain object should return false if the object is in an inconsistent state, and return true otherwise. But consistency predicates are not called explicitly by the programmer. Instead, they are automatically identified by the Fénix Framework at runtime and are executed at the end of each transaction that creates or changes any of the application’s domain objects. If any consistency predicate returns false, the domain is inconsistent and the operation aborts. Otherwise, the operation may commit.

What complicates the implementation of consistency predicates is that the consistency of one object can depend on the state of other objects. In our example, the consistency of the client depends on the state of all of its accounts. A consequence of this property is that when a transaction changes an entity (e.g., an account), it may need to verify the consistency of a different entity (e.g., the client). To allow this verification efficiently, the Fénix Framework creates and maintains a dependency network, implemented with Dependence Records.

A DependenceRecord stores the data dependencies; i.e. which objects were read on the last execution of a predicate for one object instance (the dependent), as illustrated in Figure 2. In other words, it indicates what happened the last time that the system executed the predicate for this object. Each time a transaction verifies the predicate, it will discard and rebuild the DependenceRecord, which must always be kept up-to-date.

Fig. 2: DependenceRecord model and its relations to the domain objects.
In the banking example, the DependenceRecords hold the information that the client’s consistency depends on the accounts, as seen in Figure 3. When a transaction changes an account, it uses the DependenceRecords to know that there is another entity (the client), whose consistency must be rechecked.

![Fig. 3: DependenceRecord of a Client, and the Accounts that it depends on.](image)

Thus, it is crucial to keep the DependenceRecords always up-to-date, which is the only way to keep the domain consistent, and never add inconsistencies.

The first problem with the implementation of consistency predicates in the Fénix Framework is that DependenceRecords are not persistent, unlike the application’s data. Therefore, when the application restarts, no records exist and rebuilding them from scratch is very time-consuming. A solution to this problem is to make the records persistent, like the rest of the application’s data.

Still, there is another complication: the dependence records do not take into account changes in the consistency predicates themselves. They assume that the only thing that can change is the application’s data, but that is not true when the application undergoes incremental development. As newer versions of the application are deployed, programmers may remove predicates, add new predicates, or change the code of existing predicates. Unfortunately, these changes in the predicates can make some of the persisted dependence records invalid.

3 Supporting Changes in the Consistency Predicates

In this section, we describe how we extended the Fénix Framework to keep persistent information about the consistency predicates, to detect and to handle changes made in the set of predicates of an application.

Our first goal is to detect old and new predicates as they are removed or added to classes with already existing objects. The framework must execute new predicates for the first time for those existing objects, and build the DependenceRecords that will keep them consistent according to the new predicates.

To this end, we created a set of domain entities to store information about the predicates, and the application’s domain classes. They were implemented as
classes and bidirectional relations among them in the framework’s own domain modeling language: DML.

During its initialization, the framework will create and persist objects of its new domain entities, which represent the current state of the target application’s domain code. This way, whenever a new version of code is deployed and the framework restarts, the next initialization can compare the previously persisted representation to the current state of the code. That comparison allows the framework to detect code changes.

Figure 4 shows the representation of the Fénix Framework’s domain model created to support our extension. In the following sections, we describe the role of each of the presented entities.

![Domain Model Diagram]

**Fig. 4:** The Fénix Framework’s domain model.

### 3.1 Persistent Dependence Record

The **PersistentDependenceRecord** is the persistent version of the original, transient DependenceRecord (Figure 2). It assures that its dependent object remains consistent, according to a certain consistency predicate.
The PersistentDependenceRecord keeps a relation to the KnownConsistencyPredicate, which represents the predicate that has executed. The KnownConsistencyPredicate will be described later, in Section 3.3.

The PersistentDependenceRecord is related to the dependent object for which the predicate executed, and that is being kept consistent. It is also related to all the depended objects, on which the predicate’s execution depends. Both of these relations involve the AbstractDomainObject, which is the abstract top superclass of all domain objects in the Fénix Framework.

In the banking example (Figure 3), the dependent object is the Client. The depended objects are the Accounts on which the Client’s consistency depends.

The PersistentDependenceRecord contains a new boolean slot consistent that will store the last result of the predicate execution. This value may be false for objects that already existed when a new predicate was created, and the objects were inconsistent according to the new predicate.

As we shall see, this information can be used to implement inconsistency-tolerant transactions. Anyhow, the PersistentDependenceRecords now keep track of all the objects that are inconsistent.

3.2 Meta Classes and Existing Objects

The PersistenceFenixFrameworkRoot is the singleton root class that serves as the entry point to the framework’s domain.

Each PersistentDomainMetaClass object represents one class in the target application’s domain. The domainClass slot stores the represented java.lang.Class. In the banking example, the framework will create a meta class object to represent the Client, and another for the Account.

A meta class is directly related to the PersistenceFenixFrameworkRoot. So, the framework can obtain a list of all existing meta classes from its root object.

A meta class is also related to its meta superclass, and meta subclasses. With this relation, the framework keeps the full domain class hierarchy represented.

Moreover, a meta class is related to all the existing domain objects of that class. Whenever a new object is created, the AbstractDomainObject’s constructor is invoked, which adds the object to this relation. Whenever an old object is deleted, it removes itself from this relation.

3.3 Known Consistency Predicates

The KnownConsistencyPredicate represents a consistency predicate that is known to exist in the code. The predicate slot stores the instance of java.lang.reflect.Method that implements the predicate.

The KnownConsistencyPredicate is related to the PersistentDomainMetaClass that declares it. Each meta class links to the several KnownConsistencyPredicates that its class declares.

The PersistentDependenceRecord needs to access the predicate that executed for its object. In Figure 4, the top relation between the KnownConsistencyPred-
icate and the PersistentDependenceRecord allows this access. It also gives the KnownConsistencyPredicate a list of all dependence records for that predicate.

The bottom relation between the KnownConsistencyPredicate and the PersistentDependenceRecord stores only the dependence records that are inconsistent, according to the consistent slot of the PersistentDependenceRecord. The developer can now easily obtain the inconsistent objects of a predicate.

It is important to distinguish between the different kinds of predicates supported, because they may be included inside a domain class hierarchy. A predicate in a hierarchy can override another predicate at the superclass, and be overridden by many predicates at the subclasses.

However, this situation does not apply to private predicates. For simplicity purposes, the following discussion will refer to the three kinds of predicates supported, as follows:

- **Private predicates** are those implemented with a private method. The PrivateConsistencyPredicate entity represents these methods.
- **Public predicates** are implemented with a non-final, public or protected method. The PublicConsistencyPredicate entity represents these methods.
- **Final predicates** are implemented with a final, public or protected method. The FinalConsistencyPredicate entity represents these methods.

All of these classes inherit the predicate slot and relations to the meta class and the dependence records.

Because the PublicConsistencyPredicate can be located inside a class hierarchy, the framework contains information about the overridden and the overriding predicates. This information is kept in the relation between one overridden PublicConsistencyPredicate, and several overriding PublicConsistencyPredicates. Only the PublicConsistencyPredicate and the FinalConsistencyPredicate classes include this relation, because private methods do not override each other in Java.

### 3.4 Detecting and Handling Changes to the Consistency Predicates

The first part of the framework’s initialization keeps the meta class hierarchy updated. It compares the previous PersistentDomainMetaClasses to the current domain model of the application, to detect changes to the domain class hierarchy.

The framework deletes the meta classes of old domain classes that have been removed from the code. It also deletes its subclasses, existing objects, consistency predicates, and dependence records.

The framework keeps the existing meta classes whose domain classes still exist in the code with same class hierarchy. We are concerned with updating only the existing meta classes whose superclass has changed in DML. Those metaclasses might inherit a different set of consistency predicates. The framework deletes the existing meta classes whose superclasses have changed. This deletion allows the framework to detect those classes as brand new classes afterwards.

After deleting the old and modified meta classes, the framework can start creating new meta classes. It creates one new PersistentDomainMetaClass for
each new domain class that has no associated PersistentDomainMetaClass. Then, it initializes the superclass of each created meta class.

The second part of the initialization keeps the declared consistency predicates of each meta class up-to-date. By comparing the previous KnownConsistencyPredicates to the current code of the application’s classes, the framework detects changes to the consistency predicates.

The first code change to consider is the introduction of a new predicate in a domain class. If the new predicate is public, for instance, it can simultaneously override and be overridden. It replaces the existing superclass’ predicate for its class, and for the subclasses that do not already override it.

For each new predicate that does not yet have a corresponding KnownConsistencyPredicate, the framework creates the correct KnownConsistencyPredicate to represent it, based on the method’s modifiers. Then, the framework executes the new predicates for the objects of the affected classes. The affected classes are determined by the predicate’s type, and its overriding predicates.

The second possible code change is the removal of an old predicate from a domain class. If the old predicate is public, for instance, it could override another public predicate at a superclass, and have been overridden by other predicates at the subclasses. This old public predicate was a refinement of an existing rule, that did not necessarily apply to all subclasses. Figure 5 illustrates the example of an old public predicate that used to override and be overridden.

![Diagram](image.png)

**Fig. 5:** Results of removing an old public predicate from a class hierarchy.

The framework deletes each old KnownConsistencyPredicate whose method no longer exists in the code, along with its dependence records. Moreover, as illustrated in Figure 5, the framework also obtains the overridden predicate and executes it from its class downwards.
The third and last supported code change is the change of an existing predicate’s signature (visibility or name). Consider the blue public predicate at the top superclass, that is overridden at the subclasses in Figure 5 b. The implementation of that blue predicate is only checked for objects of the superclass alone. However, if the predicate’s name is changed, or if its visibility is set to private, it will no longer be overridden at the subclasses. The framework treats predicates whose method signatures have changed like brand new predicates, and executes them for the whole class hierarchy.

4 Inconsistency Tolerance

As we said before, when a new consistency predicate is added to an application that already has existing data, it may happen that some of the data is inconsistent according to the new predicate.

Yet, if a certain existing domain object is inconsistent, the transactions that check the predicate of this object will simply abort. Aborting them will not perform several read/write operations on the domain. In turn, some part of the domain data may no longer progress in time. If an object is inconsistent, any operation attempting to modify this object will fail, until someone corrects it.

This mishap may cause users to perceive the application as being faulty, as the application’s liveness is compromised. We argue that the use of inconsistency tolerance is a way to work around this problem.

Until now, the transaction’s decision to either abort or commit is based only on the result of the predicate. If a predicate returned false (the object is inconsistent) then the transaction would abort, otherwise it would commit. We argue that this decision should also be based on the previous state of the object.

To make the transactions inconsistency tolerant, the system must make a finer grain distinction among a few types of transaction operations. In regard to a particular predicate, a transaction may:

- **Corrupt** a consistent object, by making it inconsistent.
- **Fix** an inconsistent object, by making it consistent.
- **Keep** an object **consistent**.
- **Keep** an object **inconsistent**.

Transactions might modify several objects and cause several of these operations. With these operation types, we are now able to define a new desired semantics:

*Transactions that corrupt any object should always abort.*

*Transactions that only fix or keep objects consistent should commit.*

*Transactions that only keep objects inconsistent should commit.*

In this last case, the framework should not abort these transactions, because they do not insert any new inconsistencies.
This inconsistency-tolerant semantics should be optional, because some predicates may deal with sensitive information. Only the less critical predicates should be tolerant to inconsistencies. We believe that this solution allows for a fine tradeoff between intransigent consistency and uninterrupted system liveliness.

To support this semantics, we extended the `@ConsistencyPredicate` annotation with a boolean parameter called `inconsistencyTolerant` that defaults to false. If the developer chooses to set the parameter to true, the annotated predicate will tolerate inconsistencies.

The implementation that allows this behavior needs to determine if the previous state of the object was indeed inconsistent. This information is available in the `consistent` slot of the `PersistentDependenceRecord` (Section 3.1). Therefore, the framework can use this slot both to track existing inconsistencies and to allow inconsistency tolerance.

5 Related Work

The Design by Contract approach is an object-oriented design technique [13,12] that specifies a behavior contract for each class that defines invariants. Invariants follow Hoare’s well-known axiomatic pattern [9]. They are specification rules that are defined by boolean expressions that must be true at certain points in time.

To support the checking of these rules in runtime, Meyer included an implementation in Eiffel [14] that detects flaws in the code of the application. Most existing implementations either make runtime checks for debugging purposes [4], or include formal development methods to prove the program’s correctness [10,11]. They serve the sole purpose of controlling the quality of the application code, and are usually disabled for production environment, after the testing or the proofs are complete.

And here lies one of the fundamental differences of the consistency predicates. Consistency predicates implement consistency rules that indicate if the data is consistent, rather than if the code is bug-free. They are part of the domain model’s implementation and are meant to have an active effect on the application’s runtime in a production environment. Moreover, given their purpose, invariants do not address the problems described in this paper: how to handle changes in the consistency rules and how to manage existing persistent data, which may have inconsistencies.

An approach that is quite similar to our use of consistency predicates is the use of data invariants in STM Haskell [8]. In this approach, dynamic invariants are added to the system in runtime, with the same goal of consistency predicates: prevent transactions that violate the invariants to commit. However, because invariants are added at runtime, and not at compile time, it may happen that a transaction attempting to introduce an invariant finds out that some of the existing data violates that invariant. In this case, STM Haskell simply aborts the transaction that tries to add the invariant. Thus, the transaction will discard the invariant, and will never register it. Moreover, Haskell is not an object
oriented programming language, so invariants are not placed inside classes. The consistency predicates, however, are a permanent part of the code that is never discarded. The development team can give each predicate a semantic meaningful name, and place it inside the class that is responsible for enforcing that rule.

Finally, the code detection techniques presented have some similarities with change impact analysis. Change Impact Analysis [1] is typically used to manage software code changes. It makes an analysis of the consequences of modifications within parts of the application, and presents it to the programmer.

In fact, several researchers (e.g., [16,17,18]), have proposed a few analysis and debugging tools for Java, based on impact analysis. These implementations rely on a series of tests, and trace each test’s execution to build a call graph. Each call graph allows to determine which parts of the code does each test depend on. Then, they determine the influence of code changes on each test.

These implementations are similar to the code change detection techniques presented in this work. They are concerned with the detection of new methods inside a class hierarchy, to fully support method overriding. They are also aware of changes made to a method’s modifiers, such as the visibility.

However, their purposes are again very distinct from ours. The main goal of change impact analysis is to provide the programmer with the analysis of source code changes and to help in the detection of bugs. The goal of our code detection techniques is rather to keep the data consistent in runtime.

6 Conclusion and Future Work

This work describes an extension to our previous work on consistency predicates, which deals with the typical case of an enterprise application that has persistent data and is developed in an incremental way. Our solution maintains the simple programming API for the programmer of the Fénix Framework, who may add or remove predicates freely. The framework detects these changes and maintains the dependency network updated to prevent new inconsistencies. Moreover, the framework provides structured information to allow the programmer to access existing inconsistent objects with ease.

We have also proposed a new semantics of inconsistency tolerance for the consistency predicates, to deal gracefully with the case of existing data that is known to be inconsistent after it has been created. Inconsistency tolerance allows the system operations to modify already inconsistent objects freely, especially if these objects do not hold sensitive information.

Both of these contributions were implemented in the Fénix Framework, and a more detailed description of their design, implementation, and validation is available in [15].

One limitation of our current implementation, however, is that it does not detect changes to the code implementing an existing predicate. To address this problem, as future work, we propose to enable the detection of method code changes, which can be achieved by creating a domain entity to represent regular domain methods, much like what was done with the KnownConsistencyPredicate.
This way, the framework could detect changes in the persisted domain methods, and determine which predicates to reexecute.

Another interesting topic for future work is to allow the framework to perform automatic inconsistency fixes. This task could be implemented by associating a fixing method with each consistency predicate. Then, whenever the framework detects, or tolerates, an inconsistency, it could attempt to fix that inconsistency.

References