Diagnosis and semi-automatic correction of detected design inconsistencies in source code

Sergio Castro Johan Brichau Kim Mens
Université catholique de Louvain, Belgium
{sergio.castro, johan.brichau, kim.mens}@uclouvain.be

Abstract

In order to alleviate design decay, different program design documentation techniques are used for the specification and detection of design inconsistencies in code. However, these design documentation techniques do not always provide support for the diagnosis and (semi-) automatic correction of such inconsistencies. In case they do, corrective solutions are typically targeted to a reduced set of pre-defined inconsistency problems, and they are not easily customizable to new kinds of consistency checks defined by a user. In particular, they cannot infer possible corrective actions to solve new user-defined inconsistency problems. In this paper, we present a technique for the diagnosis and (semi-) automatic correction of inconsistencies in the context of an existing tool for inconsistency management: IntensiVE. Our technique uses logic abductive reasoning to infer solutions to detected user-defined inconsistencies, starting from basic composable *corrective actions*. A first prototype implementing our technique on top of IntensiVE is shown.

Keywords Design inconsistency, diagnosis, correction, abductive reasoning , IntensiVE, SOUL, Smalltalk

1. Introduction

The lack of traceability of program design decisions (e.g., *coding conventions* [Beck 1997], *idioms* [Coplien 1992], *design patterns* [Gamma et al. 1995], *design regularities* [Minsky 1996]), and lack of documentation and verification techniques that can enforce these decisions, cause a continuous erosion of the design of a software system [van Gurp and Bosch 2001]. Therefore, program design documentation techniques are needed to provide developers with a complete understanding of the design decisions that govern the implementation. In this way, they can make efficient modifications to such implementation when needed [Letovsky and Soloway 1986] [Robillard et al. 2004], while avoiding changes that do not respect existing design (i.e., create design inconsistencies) [Parnas 1994]. This improves the overall quality of the code in terms of reusability, extensibility and comprehensibility. Though some tools provide for the detection and even correction of certain kinds of design inconsistencies in code, such as stylistic conventions and some bad practices (e.g., CheckStyle [Che06] and Lint [Johnson 2007]), these tools do not allow for a general means of checking a broad number of *regularities* [Minsky and Minsky 1994] that need to be defined in a real software system, nor do they offer a mechanism for automatically inferring a corrective solution for user-defined inconsistency problems.

In this paper, we present a first prototype implementation of a technique based on logic abductive reasoning that overcomes some of these limitations. This prototype is implemented on top of IntensiVE [Mens et al. 2005, Kellens 2007], a tool suite for specifying and enforcing a wide variety of structural regularities in the source code of a system. Software engineers can define regularities by means of source-code queries that gather specific source-code entities into *intensional views*, upon which constraints are imposed. With IntensiVE, developers can specify the regularities they deem interesting and invoke their detection at any time they desire. Upon such detection, violations of the regularities in the source code or in the design documentation are reported by the tool suite.

However, the current support offered by IntensiVE for inconsistency management is limited to *specification* and *verification* of structural regularities. In this work we attempt to expand this support with:

1. a *diagnosis* mechanism that provides a programmer with possible explanations of the causes that created an inconsistency;
2. a semi-automatic *correction* mechanism that supports the programmer in the task of applying corrective actions on code, in such a way that the system will be more consistent.

ESUG '09 August 31, Brest

As explained in the next sections, these two objectives will be reached using abductive reasoning techniques [Pierce 1935].

This paper is structured as follows. Section 2 provides an overview of how IntensiVE is currently used for the specification and detection of inconsistencies in code. Section 3 explains why the notion of logical abduction is suitable for the diagnosis and correction of inconsistencies. Section 4 introduces our abductive reasoning framework for inconsistency management with the help of a small example. Section 5 shows with a less trivial example our technique, and explains how existing corrective actions can be reused in new contexts. Section 6 presents related work that makes use of abductive techniques for inconsistency management in software engineering. Section 7 discusses our future work and finally section 8 presents our conclusions.

2. Using IntensiVE for the specification and verification of regularities

In this section we show with a small example how IntensiVE can be used for specifying and detecting a common bug in code. In section 5 we will show on a less trivial example how IntensiVE could verify the correct implementation of a design pattern.

Inspired by the well known *code critic checks* [Johnson 2007] provided by many developing tools (e.g., VisualWorks Smalltalk [Cincom Systems, Inc. 2007]), we have selected as a first example one typical check: the verification that any class implementing the binary message = also provides an implementation for the method *hash*, and vice versa.

Although this check is already provided by Smalltalk, we have decided to show here how this can be implemented in IntensiVE in order to demonstrate how *reusable* (as we will see in section 5) and *executable* corrective actions can be defined. Something that is not currently provided by existing development tools.

2.1 Definition of an intensional view

An intensional view represents a set of source-code entities (e.g., methods, classes, etc) that make up the implementation of a concept of interest. In many cases, source code entities implementing a same concept share a structural property. For example: all accessor methods share the coding convention that their names should be the same that the instance variables they are accessing.

More precisely, an intensional view is an intensionally specified *set of tuples* of source-code entities. The components of the tuple are the source code elements relevant to the concept the intensional view is expressing (e.g., the extension of an intensional view describing all the accessors in the classes belonging to a particular package, is a set of tuples containing each of them only one element: an accessor method). The size of this tuple and the code entities it contains, is part of the definition of the intensional view.

A relevant property of intensional views is that their sets of tuples are not defined by enumeration (i.e., *extensionally*) but by means of an *intension*. Similar to set theory, an intension is a descriptive definition that yields, upon evaluation, the set of tuples of entities belonging to the extension of the view. Although IntensiVE is independent of the query language used, our tool tightly integrates with the logic (meta-) programming language SOUL[Wuyts 2001] (a dialect of Prolog [Shapiro and Sterling 1994]).

Figure 1 shows the definition of an Intensional View in IntensiVE that groups all the classes implementing a *hash* method. The SOUL query used to gather these classes is:

```
methodWithNameInClass(?,hash,?class)
```

Upon execution, this query will bind the tuple variable *?class* to classes implementing methods with name *hash*.

Grouping elements belonging to a same concept is a very useful mechanism for code exploration and comprehension. However, being able to define constraints over the elements belonging to these concepts is what makes Intensional Views particularly suitable for the specification and detection of inconsistencies. Different mechanisms are possible in IntensiVE for specifying such constraints, but for the sake of simplicity we will talk only about the mechanism of *alternative views*.

An alternative view is an alternative definition of an existing concept, where all the source code elements in an alternative definition should also be part of the elements of the original concept and vice versa. For example, we can say that the same groups of classes redefining a *hash* method, should also redefine the = method. Then this view is composed of two alternatives. The second alternative uses the following SOUL query to bind the tuple variable *?class* to all the classes implementing methods with name = :

```
methodWithNameInClass(?,=,?class)
```

2.2 Detection of inconsistencies

Once multiple alternatives have been defined for a view, it is possible to accomplish a verification that will report a list with all the source elements that are present in both alternatives (consistencies) and source code elements that are defined in only one of the alternatives (inconsistencies).

Figure 2 shows a verification of the previously described Intensional View with two alternatives. The third column shows classes implementing the = method (bullets without 'X'). At the second column we can see classes implementing the *hash* method. A row with red bullets (with 'X') means that the view does not satisfy one of the two alternative descriptions of the view, for example a class that implements the *hash* method and not the =, or vice versa.

3. Abduction as a diagnosis mechanism

At this point we have seen how IntensiVE can report detected inconsistencies between design and code (for example, the adherence of the code to the design regularity that

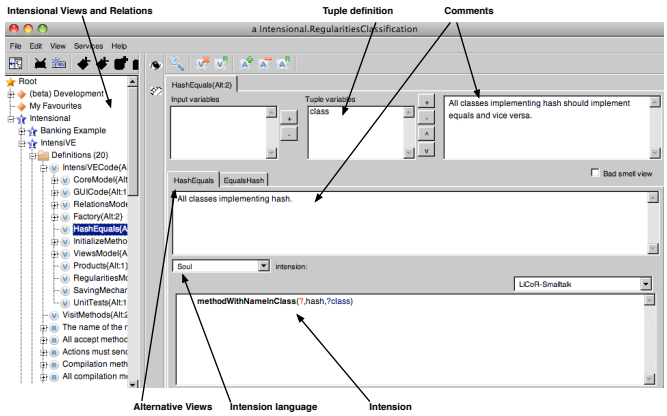

Figure 1. A view with two alternatives. The first alternative queries classes implementing the *hash* method.

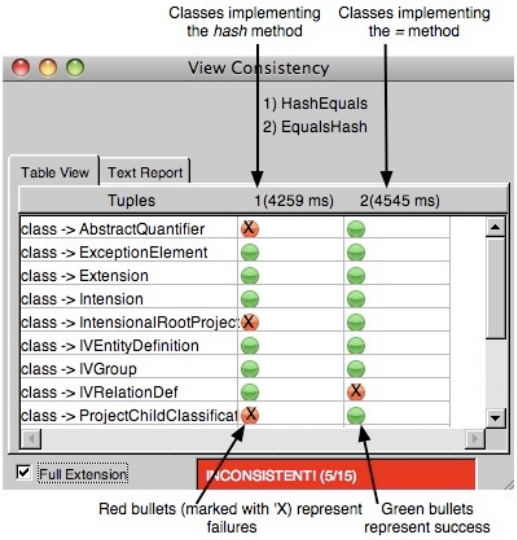

Figure 2. Consistency check of the *hash* versus = methods regularity.

every class implementing a *hash* method should implement an = method and vice versa). Before being able to semi-automatically correct detected problems, we need a mechanism to *diagnose* which are the possible causes of these problems (i.e., hypothesizing which are the properties of the source code elements that could be causing the inconsistency [Reiter 1987]).

Since in our diagnosis problem the description of a system is already expressed with logic terms [Wuyts 1998, Mens and Kellens 2006], being pragmatics we decided to adopt a diagnosis strategy based on logic programming too. This decision led us directly to *abductive reasoning* [Flach 1994].

Pierce was the first to formally introduce the notion of *abduction* as one of the three fundamentals forms of human reasoning (the other two being *deduction* and *induction*) [Pierce 1935].

Abductive reasoning plays the role of a generator of new ideas or hypotheses [Yu 1994]. It has been shown to be specially useful as an inference process for scientific discovery of explanatory hypotheses for anomalous phenomena [Paavola 2004, Pierce 1955] and, more specific to our problem, as a mechanism for the detection and diagnosis of inconsistencies in different fields of software engineering [Russo and Nuseibeh 2000, Nuseibeh and Russo 2007, Russo et al. 2000, Satoh 1998b, 2000, Denecker and Kakas 1993].

Intuitively, with abduction we can find a set of hypotheses (or explanations) that, when added to a given formal specification, will allow an observation to be inferred [Russo and Nuseibeh 2000]. A more formal definition is provided in [Denecker and Kakas 1993]. A typical *Abductive Logic Programming Theory* is defined as a triple (P, A, IC), where P is a logic program (a set of Horn clauses), A is a set of *abducible* atoms and IC is a set of logical formulas referred as integrity constraints.

Given such a logic theory (P, A, IC), an abductive explanation for a query Q is a set $\Delta \subseteq A$ of ground atoms[1] such that:

- $P \cup \Delta \models Q$
- $P \cup \Delta \models IC$
- $P \cup \Delta$ is consistent

That is, any explanation Δ together with the original theory P should satisfy the observation Q, should respect the integrity constraints IC, and should be consistent. Note that the set Δ should satisfy the criteria of: (1) belonging to the set of abducible predicates A and (2) it should be minimal, in the sense that no abductive explanation is subsumed by another abductive explanation [Russo and Nuseibeh 2000].

We will illustrate abductive reasoning with a small example adapted from [Flach 1994] and [Inoue and Sakama

[1] Abduction produces only ground explanations. The inference of new rules, or non-ground explanations belongs to the domain of induction

1998]. Assuming no special integrity constraints, we could define our abductive logic programming theory (P, A, IC) as:

- P is the logic program:

```
flies(?x) if bird(?x), not(ab(?x))
ab(?x) if broken-wing(?x)
broken-wing(tweety)
```

- A is the set {*bird(?x), broken-wing(?x)*}.
- IC is an empty set {}.

Then if Q is the observation: *flies(opus)*, we find that Δ is the set {*bird(opus)*}.

Explanation: in order for the query *flies(opus)* to succeed, both *bird(opus)* and *not ab(opus)* should succeed. Though there is not any fact explicitly stating that *opus* is a bird, the predicate *bird(?x)* is abducible, so we know that a possible explanation for the observation *flies(opus)* is that *opus* is a bird. Since *not(ab(opus))* succeeds, no additional explanation is required.

In addition to the generation of such facts that when added to a theory make a query succeed, useful generalizations to the concept of abduction allow the retracting of facts (referred as *negative explanations*) and *un*explanations of negative observations (referred as *anti-explanations*) [Inoue and Sakama 1998].

To understand the intuition behind this, let us revisit our previous example. If we ask for an explanation of the query *flies(tweety)*, normally abduction would fail in finding such explanation. We recall that for the query *flies(tweety)* to succeed, both *bird(tweety)* and *not(ab(tweety))* should succeed. Though the first predicate can be abduced (i.e., *bird(?x)* is abducible), a complete explanation cannot be found since *not(ab(tweety))* fails and no explanation can be found to make it succeed. This is because *ab(tweety)* succeeds giving that *broken-wing(tweety)* also succeeds. With the notion of negative explanations, we could say that *broken-wing(tweety)* is a negative explanation of *flies(tweety)*. In other words, to make *flies(tweety)* succeed, we have to retract from our theory the fact that tweety has a broken wing (i.e., to make tweety fly again we have to heal its wing).

In the next section we demonstrate how abduction can be used as an effective mechanism for diagnosing and correcting design inconsistency problems in source code.

4. Diagnosing and correcting inconsistencies

4.1 The diagnosis problem in IntensiVE

Reiter [Reiter 1987] defines the diagnosis problem as the search for the components of a system that under the assumption they are faulty, will explain an inconsistency between the description of that system and certain anomalous observations .

In our context, we consider the description of a system as the design regularities over source code that govern an implementation. Our observations are the source code elements that fulfill or not these design regularities. In other words, our problem consists in determining which are the faulty source code elements that are causing design inconsistencies to occur, and how we can repair these source code elements in order to fix those inconsistencies.

Revisiting our example of section 2, figure 2 shows a comparison between two sets of source code elements that should be identical: the set of classes defining a *hash* method and the set of classes defining an = method. In our example, an inconsistency is found if a class is present in the first set (i.e., it implements the method *hash*) and not in the second (i.e., it does not implement the method =) or vice versa.

If a class is not present in the first set, for example: *AbstractQuantifier*, it is because the result of the query gathering the tuples belonging to this set is not including this class. In other words, the query *methodWithNameInClass(?,hash,[AbstractQuantifier])* is failing.

Even if we do not know how this query is defined in SOUL, intuitively we know that if we would add a method *hash* to *AbstractQuantifier*, then the result of the query would include the class *AbstractQuantifier*.

An alternative way to resolve this inconsistency would have been to retract the class *AbstractQuantifier* from the second set. This is because if a class is not present in the first nor the second set, no inconsistency will be found. The class *AbstractQuantifier* is present in the second set since the query *methodWithNameInClass(?,=,[AbstractQuantifier])* is succeeding. Intuitively, we know that if we delete the method = from *AbstractQuantifier*, the result of the query will not include anymore the class *AbstractQuantifier*.

In general, in order to solve inconsistencies present in views with two (or more) alternatives, for every inconsistency found a programmer must choose between one of these two options:

- Find a code transformation that will add tuples to the sets where they are missing, giving that these tuples are present in the sets of the other alternatives.
- Find a code transformation that will retract tuples from the sets where they should not be, giving that these tuples are not present in the sets of the other alternatives.

Therefore, we found that a programmer willing to diagnose and correct this kind of inconsistencies, is faced with these problems:

- Choose the sets to fix, either adding or retracting tuples from these sets.
- Find explanations why a tuple is not present in the set chosen (in case a tuple needs to be added) or why it is present (in case a tuple needs to be retracted).
- Finding, performing and verifying the right code transformation that will correct the inconsistency.

The first two questions are related to diagnosis, the third question is related to correction of inconsistencies. In this section we explain how using abductive reasoning techniques, it is possible to partially solve these problems.

4.2 A framework for diagnosis and correcting inconsistencies

We have intuitively described the process of diagnosing and correcting inconsistencies in Intensional Views with two or more alternatives. In this section we propose a more methodological mechanism using abductive reasoning.

As mentioned in section 3, in order to use abduction for diagnosis we have to predefine a subset of sentences that can be part of an explaining answer, called abducibles. In abductive reasoning, these sentences should not appear as consequence of rules [Flach 1994, Denecker and Kakas 1993, Russo and Nuseibeh 2000], in order to avoid the generation of explanations that themselves can be explained in terms of other sentences.

However, we relax this constraint in our technique. In SOUL, the structure of an object-oriented program is *reified* (i.e., given an equivalent representation of its structure in a logic language) using queries over an abstract syntax tree. These queries intensionally represent the logic facts that define the structure of a program. Therefore, in SOUL there is not an explicit set of final clauses (or facts) expressing the structure of a program [Wuyts 2001, 1998] that we could define as abducibles. Instead, we have defined any predicate stating a structural relationship among source code elements, and for which a corrective action can be defined, as potentially abducible.

Since abducible predicates have to be explicitly declared, our framework has the notion of *corrective actions*. These are a set of clauses following this pattern:

correct(⟨*abduciblePredicate*⟩, ⟨*action*⟩, ⟨*description*⟩).

The meaning of the predicate parameters is

abduciblePredicate: an abducible predicate. All the possible explanations are expressed in terms of predicates declared as abducible. Two possible types of abducibles can be declared: abducibles for explanations (denoted by the *in* functor) and abducibles for anti-explanations (denoted by the *out* functor). In other words: *in*-abducibles, denoting explanations, declare that a predicate will be added to the theory after the execution of an associated corrective action. *out*-abducibles, denoting anti-explanations, declare that a predicate will be retracted from the theory after the execution of an associated corrective action. This terminology is taken from [Inoue and Sakama 1998].

action: a code transformation procedure that will apply a corrective action associated with an explanation (i.e., addition of the predicate *abduciblePredicate* to the theory) or anti-explanation (i.e., retraction of the predicate *abduciblePredicate* from the theory).

description: a textual description of an explanation and its associated corrective action.

For example, the following is the abducible predicate for *methodInClass(?method,?class)* with an associated anti-explanation:

```
correct(out(methodInClass(-?method,+?class)),
[[(Refactory.Browser.RemoveMethodRefactoring removeMethods:
        (OrderedCollection with: ?method selector)
        from: ?class) execute.]]),
['this corrective action deletes the method ',
        (?method selector asString),
        ' in class ', (?class asString)]
```

As we mentioned in section 3, the anti-explanation of an observation is a set of facts that need to be added (*positive anti-explanation*) or retracted (*negative anti-explanation*) from our *theory* (i.e., the reification in logic of an object oriented program) in order to *un*explain [Inoue and Sakama 1998] an observation. However, we mentioned that in SOUL an object oriented program is not reified as a set of logic facts, but as queries over its parse tree that represent those facts. Then, instead of adding or retracting facts, an explanation/anti-explanation will be associated with a source code transformation that will change the parse tree of a program. As a result, SOUL will reify the modified parse tree, having the same effect than explicitly adding or retracting facts as it is done in traditional abduction.

In our example, an anti-explanation for the predicate *methodInClass(?method,?class)*, is associated with the removal of the method *?method* from the class *?class* (i.e., if we remove the method *?method* from *?class*, the fact *methodInClass(?method,?class)* is retracted from our theory).

We have chosen the *methodInClass/2* predicate as abducible, since the queries defining the alternatives of our Intensional View make use of it. This is because *methodWithNameInClass/3*, the SOUL query used with different parameters in the two alternatives, is defined as:

```
methodWithNameInClass( ?method, ?selector, ?class) if
        methodInClass( ?method, ?class),
        methodWithName( ?method, ?selector)
```

Once we have defined an anti-explanation for an abducible predicate, we are implicitly defining a possible anti-explanation for all the predicates that depend on the success of our abducible predicate. In other words, if the success of *methodWithNameInClass/3* depends on the success of *methodInClass/2*, an anti-explanation for *methodInClass/2* will be an anti-explanation for *methodWithNameInClass/3*.

Figure 3 shows a user choosing in IntensiVE a tuple to correct with a contextual menu. Note that the selected tuple belongs to the second alternative of our example, and that the only source code element in the tuple is the class *AbstractQuantifier*. The tool infers that the user is asking for an explanation of the possible actions the programmer could execute in order that the query *methodWithNameIn-*

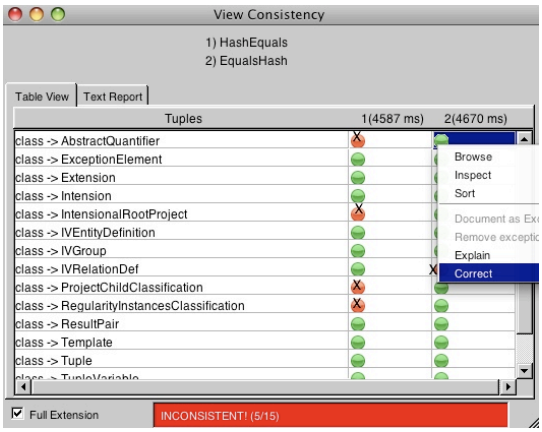

Figure 3. Choosing a tuple to correct

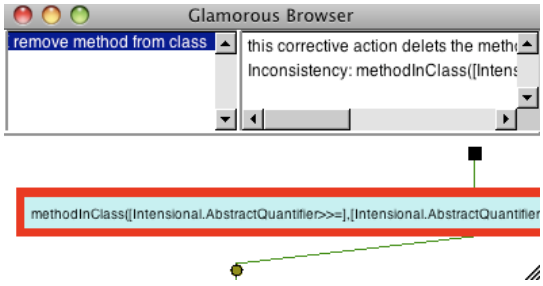

Figure 4. Corrective actions browser

Class(?,=,[AbstractQuantifier]) fails and the view becomes consistent.

Since *methodWithNameInClass/3* is defined in terms of *methodInClass/2*, when the query *methodWithNameInClass(?,=,[AbstractQuantifier])* is launched, it will be resolved to *methodInClass(?method,[AbstractQuantifier]), methodWithName(?method,=)*. The normal resolution process continues until the query succeeds, but in the meanwhile our abductive interpreter detected that *methodInClass/2* was invoked and that its execution was successful. Since we asked to the abductive interpreter for explanations to the fact that *methodWithNameInClass/3* is false, given that *methodInClass/2* is an abducible predicate with an anti-explanation associated with it, the interpreter answers that the desired result will be obtained if the fact *methodInClass/2* is retracted (i.e., the corrective action associated with *out(methodInClass/2)* is executed). The results are shown to the user in a browser together with other possible solutions. Figure 4 shows an initial version of this browser developed with Glamour [Bunge 2009], from where the programmer can easily inspect a solution and execute its associated code transformation.

The top left panel shows a list of possible solutions, the top right panel a textual description of this solution and the bottom panel shows a visual explanation of the execution of the logic abductive interpreter. Note that only one solution is shown, since only one relevant abducible predicate was defined in our example.

Although it can be seen as if our intension was just to define a specific solution to the problem of classes with an unnecessary method =, in fact we have defined a corrective action that can be reused where an anti-explanation for a predicate using *methodInClass/2* is needed. For example, if in the consistency check (figure 3) the programmer would have chosen to retract the class *IVRelationDef* from the first set of the view instead (i.e., the classes implementing the method *hash*), no additional work would be needed, since the query *methodWithNameInClass(?,hash,[IVRelationDef])* is also defined in terms of *methodInClass/2*.

5. Example: the command design pattern

5.1 Description of the problem

In this section we use our framework for a less trivial example. We have taken a regularity from an existing Smalltalk project: the *StarBrowser* [Wuyts and Ducasse 2003]. This project implements a well-known design pattern: the *command design pattern* [Gamma et al. 1995]. In the StarBrowser implementation of this pattern, command objects are instances of the *AbstractAction* class. They should override the method *perform* to implement the action to execute. An interesting feature of action classes is that some of them are undoable. As long as an *undoAction* method is provided, the action performed can be undone. Furthermore, a design constraint requires all the classes which actions can be undone, to return *true* in their method *isUndoable*. Otherwise, the class will be considered as not undoable. Similarly, any class returning *true* in the method *isUndoable* and not providing an implementation for the method *undoAction* is violating our consistency rule.

Figure 5 illustrates various examples of classes that should adhere to this design constraint. The default implementation of *isUndoable* in *AbstractAction* is returning *false*. Any class providing an *undoAction* method should override the method *isUndoable* returning *true*.

We can see in the figure that the class *DummyAction* is consistent according to our design, since an *undoAction* method is provided and the *isUndoable* method is returning *true*. However, this is not the case for *ExperimentalAction* and *DrawAction*. *ExperimentalAction* implements an *undoAction* method, but it does not override the method *isUndoable*. Since this method is returning *false* in its parent class, an inconsistency is produced because the class is answering that it is not an undoable class, but it provides undo behaviour.

DrawAction is overriding the method *isUndoable* returning true, but no implementation for undo behaviour is provided (i.e., the class does not implement an *undoAction* method).

Finally, the class *QueryAction* provides an implementation of *undoAction*, but returns the result of a block in the method *isUndoable*. Therefore, knowing for sure if the class is following or not our design rule is often impossible to know with certainty.

In the remainder of this section we show how we can verify this design pattern using IntensiVE, and demonstrate how problems can be diagnosed and corrected using the abductive reasoning techniques shown in this paper.

5.2 Specification and detection of inconsistencies

We start by defining an intensional view for the undoable actions. This view has two alternatives. The first alternative queries the classes implementing the method *isUndoable* and returning *true* in it. The query for this alternative is:

```
classBelow(?class,[AbstractAction]),
classChainReallyUnderstandsMethodWithName(
        ?class,?method,isUndoable),
methodReturnsBoolean(?method,[true])
```

Upon execution of this query, *classBelow/2* will bind the variable *?class* to all classes inheriting from *AbstractAction*. *classChainReallyUnderstandsMethodWithName/3* will filter this result, keeping only those classes which understand a method named *isUndoable*. Here, the variable *?method* will be bound to the *isUndoable* method. The third predicate will filter even more the results, keeping only those classes that return *true* in their *isUndoable* methods.

The query for the second alternative of the view is the following:

```
classBelow(?class,[AbstractAction]),
classChainReallyUnderstandsMethodWithName(
        ?class,?,undoAction)
```

Similarly to the first alternative, the second alternative queries the classes inheriting from *AbstractAction*. After that, the result is filtered to those classes that understand the *undoAction* method.

Upon checking these regularities in IntensiVE, we will compare the sets of classes returning *true* in the method *isUndoable*, and the set of classes providing an implementation for the method *undoAction*.

Figure 6 shows the result of this check. We can see that only few classes do not respect this design constraint. Our next step is to diagnose and correct these inconsistencies.

5.3 Diagnosis and correction of inconsistencies

In order to semi-automatically solve design inconsistency problems with our abductive reasoning technique, we presuppose the existence of a library of basic corrective actions (i.e., abducible predicates associated with one or more code transformation actions), that are reusable across different inconsistency problems. In the trivial example shown in the previous section, we illustrated the use of anti-explanations for retracting facts from our theory (i.e., *out*-abducibles). In this example, we will make use of an explanation for adding

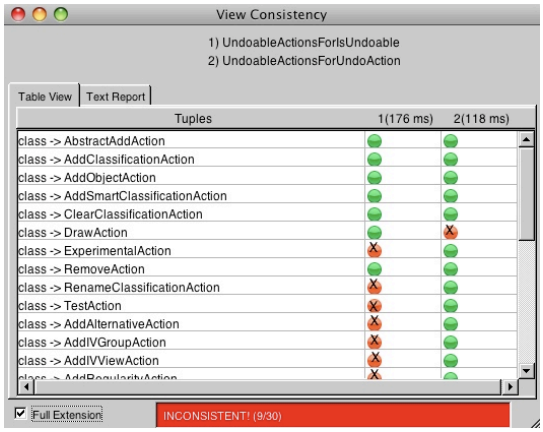

Figure 6. Consistency checking for the command design pattern in the StarBrowser.

facts to our theory (i.e., *in*-abducibles). For this, we will define a corrective action for *methodReturnsBoolean/2*.

The following corrective action declaration associates the *in*-abducible for the predicate *methodReturnsBoolean/2*, to a code transformation.

```
correct(in(methodReturnsBoolean(+?method,+?boolean)),
[[?class compile: (?selector asString,
        ' ^', ?boolean asString).]]) if
methodWithNameInClass(?method,?selector,?class),
['this solution changes the code of the method ',
        ?method asString, '.
        The new code just has a simple ',
        ?boolean asString ,' return statement']
```

We explained in section 4 that code transformations linked to *in*-abducibles should guarantee after their performance the truth of the abducible predicate (i.e., the addition of the predicate to the theory). In this case, the predicate states that a method *?method* is returning a boolean with value *?boolean*. A possible solution is creating a new method (or replacing the code of a method if it already exists) with the appropriate boolean value (taken from the value of *?boolean*). Note that this is a destructive operation that needs to be managed with extreme care, and this is the main reason that our approach is semi-automatic. It is the tool user's responsibility to make that change or not, and face the consequences if things go wrong. We envision as future work the inclusion of a transaction mechanism to rollback such destructive changes if needed.

This transformation is shown in the corrective action declaration as a Smalltalk block that will be executed when the tool user requests to trigger this corrective action.

Making use of a library of corrective actions, a programmer could ask for correcting the absence of the class *ExperimentalAction* from the first alternative of our view. This is shown in figure 7.

This user request is interpreted by our tool, as the search for an explanation of the reason why the following query is failing:

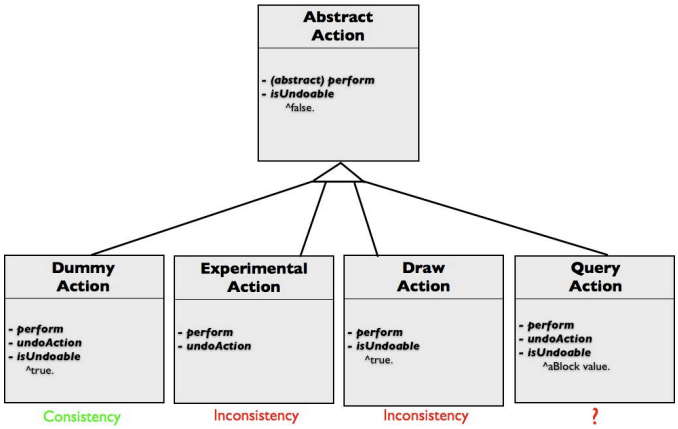

Figure 5. The command design pattern as implemented in the Starbrowser.

Figure 7. Correcting the absence of *ExperimentalAction* from the first alternative

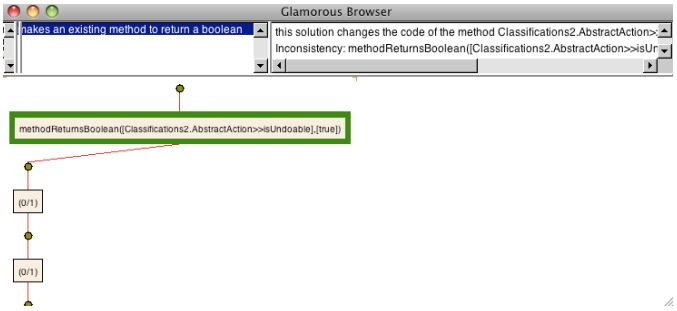

Figure 8. Corrective actions for adding *ExperimentalAction* to the first alternative of the view.

```
classBelow([ExperimentalAction],[AbstractAction]),
classChainReallyUnderstandsMethodWithName(
        [ExperimentalAction],?method,isUndoable),
methodReturnsBoolean(?method,[true])
```

The two first predicates are successful, but the last one fails, since *ExperimentalAction* is not implementing the method *isUndoable* returning *true*. However, our abductive interpreter will detect that the failed predicate *methodReturnsBoolean/2* is abducible, and that a corrective action that will add it to the theory is available. The available corrective action is shown to the user in figure 8. Upon execution of this corrective action by the user, an *isUndoable* method returning *true* will be added to the class *ExperimentalAction*, overriding the default behaviour defined in *AbstractAction*, and the inconsistency will be solved.

As a last example, let us take a look at another class. If a programmer is interested in correcting the presence of *DrawAction* in the first alternative of the view (figure 9), this request will be understood as the explanation of why the query below is succeeding:

```
classBelow([DrawAction],[AbstractAction]),
classChainReallyUnderstandsMethodWithName(
        [DrawAction],?method,isUndoable),
methodReturnsBoolean(?method,[true])
```

The three predicates of the query succeed. However, as part of the resolution process of the first predicate, *methodInClass/2* is queried (though we will not show the complete resolution tree in order to avoid unnecessary complexity). At that moment, the abductive interpreter discovers an anti-explanation for *methodInClass/2* (developed for our first example in section 4). Then, if it exists an anti-explanation for *methodInClass/2*, and *classChainReallyUnderstandsMethodWithName/3* is defined in terms of *methodInClass/2*, then the anti-explanation of *methodInClass/2* is also useful for *un*explaining *classChainReallyUnderstandsMethodWithName/3*. Then, there already exists in our framework a code transformation procedure that will make *classChainReallyUnderstandsMethodWithName([DrawAction],?,undoAction)* fails, and no new solution needs to be added to the library of corrective actions.

As we described in the previous section, the only anti-explanation available in our library of corrective actions for the predicate *methodInClass/2*, is to delete the method in the first parameter in the predicate from the class in the second parameter. Although simple, in our problem this solution can be absolutely valid, because if this method is deleted, the *DrawAction* class will use the default implementation of *isUndoable* present in its base class *AbstractAction*, which

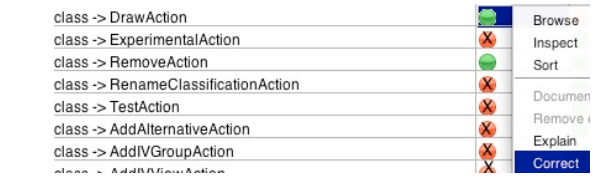

Figure 9. Correcting the existence of *DrawAction* in the first alternative

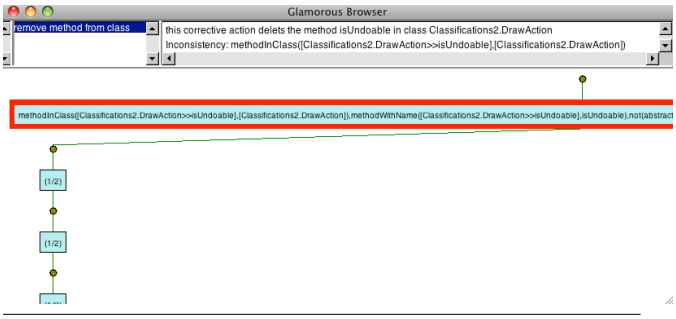

Figure 10. Corrective actions for adding *DrawAction* to the first alternative of the view.

returns *false*. That value is consistent with the fact that no *undoAction* method is provided in the class *DrawAction*.

The browser with the visualization of this solution is shown in figure 10, note the similarities with figure 4, changing only the parameters of the source code transformation.

As such, we have demonstrated that small solutions from primitive problems can be easily reused in other contexts than the original where they were needed for first time. Also, we have shown that these solutions can be relevant and useful.

Given that the examples we have explored here are still rather small and simplistic, we will see in section 7 that one of the paths in our future work is to test our technique in bigger and more realistic examples.

6. Related Work

Inconsistency management using abductive-logic programming techniques has been discussed as a promising mechanism in many areas of software engineering, such as: requirements engineering, software design, and databases among others.

Russo et al. present in [Russo and Nuseibeh 2000] a comprehensive survey on this topic. Most of the works discussed there are centered around inconsistency management in requirements engineering [Nuseibeh and Easterbrook 2000], particularly the analysis and revision of specifications [Menzies 1996, Nuseibeh and Russo 1999, Russo et al. 2000, Satoh 1998a,b, 2000].

Analysis of specifications is the analysis on the model of a system. This model is built in terms of the interaction of a system with the environment and the user-goals that this system should implement [Zave and Jackson 1997]. In this context, Inconsistency detection consists of the task of verifying system properties over such a model. It is described that in general with abduction is possible to verify, for any property $P(X)$, if the goal $P(X) \wedge \neg P(X)$ can be inferred from a specification. If that is the case an inconsistency exists, and the explanation of this goal generated for the abductive process constitutes a diagnosis of why the specification is inconsistent.

Alternatively, in [Russo et al. 2000] inconsistencies are detected and diagnosed trying to identify through abduction counter-examples of all the invariants in a system. If the abductive reasoning mechanism fails to find an answer, this establishes the validity of the invariant with respect to a system description.

Also in the work of Russo et al., inconsistency management techniques are divided according to the consistency assumptions they make about the model to examine. For example, the work in [Nuseibeh and Russo 1999] describes a specification as a composition of multiple partial specifications with or without logical inconsistencies. The approach identifies changes that address some specification inconsistencies, while leaving others. The abductive reasoning mechanism identifies evolutionary changes to perform on the specification, such that a particular consistency rule is no longer violated. Quasi-classical logic [Hunter and Nuseibeh 1998] is used as a mechanism for reasoning in a inconsistency system without *trivialization* [Besnard and Hunter 1995].

A different perspective is described in [Satoh 1998b, 2000, 1998a]. These proposals are focused on analyzing the impact of changes in an initially consistent specification. The specification revision process is in charge of re-establishing consistency in order to accommodate a given change request, and abduction in this case is used to identify additional changes on the given specification so as to re-establish consistency.

In [Satoh 1998a] Satoh describes a logic approach based on abductive reasoning for adding and deleting *pollution markers* [Balzer 1991] from a given specification in order to manage consistency after a change has been performed. The objective of pollution markers is considering inconsistencies as exceptions that can be isolated from the rest of the data.

In the domain of software modelling, Andrea et al. discuss in [Zisman and Kozlenkov 2001] a mechanism for checking inconsistencies in UML specifications. Mapping UML specifications in XMI format, they use abduction for declaring as consistency properties certain goals that should succeed following a particular course of events. If a goal do not succeed or its proof follows another course of events that the one expected, changes set indicating which axioms should be deleted or added are produced by abductive reasoning.

7. Future work

We have visioned two main venues of future work that are strongly related among them: the consideration of integrity constraints and the development of heuristic strategies for filtering distinct diagnosis solutions.

7.1 Consideration of Integrity constraints

In our abductive interpreter we have not yet considered the existence of integrity constraints. The result of an abductive interpreter could be refined with the use of these constraints [Kakas and Tony 1998]. In case they are present, abductive reasoning should generate only explanations that satisfy these constraints, and the abductive interpreter should check their satisfiabilty each time a new explanation is produced [Russo and Nuseibeh 2000]. Possible integrity constraints that we will further explore are the verification that any code transformation, in addition to solve a local inconsistency, will preserve the consistencies in all the previously consistent Intensional Views and the relations among them.

7.2 Heuristic strategies

Even though integrity constraints could be a good mechanism for filtering a certain number of wrong solutions, the process of abductive reasoning must be complemented with a further quality criterion over candidate solutions, in order to infer only the best explanations for an inconsistency[Josephson and Josephson 1994]. These additional validity considerations are not part of the abductive process itself, but complement it [Paavola 2004, Sullivan 1991]. Abductive exploration just performs the function of a model builder that should be followed for a confirmatory data analysis. Therefore, abduction plays the role of an explorer of viable paths, but further inquiry should determine which are the most plausible solutions [Yu 1994].

7.3 Development of bigger and more realistic examples

We have developed until now rather small examples, and explored how corrective actions targeted to one particular problem can be reused in other problems with different contexts. One of our immediate next steps is the definition of a more realistic case study that leads us to explore all the possibilities of our technique and its limitations.

8. Conclusions

In this work we have explored abductive logic reasoning as a promising mechanism for the diagnosis and correction of design inconsistencies in program source code. We have demonstrated, using two small examples, how to identify source code artifacts that could be causing inconsistencies, and how to associate corrective actions to them that upon execution will restore the consistency of the program source code. Furthermore, our corrective actions can be reused across distinct problems, since they are defined in terms of basic logic predicates.

References

Robert Balzer. Tolerating inconsistency. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press. ISBN 0897913914. URL `http://portal.acm.org/citation.cfm?id=256748`.

K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

Philippe Besnard and Anthony Hunter. Quasi-classical logic: Non-trivializable classical reasoning from incosistent information. In *ECSQARU '95: Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 44–51, London, UK, 1995. Springer-Verlag. ISBN 3-540-60112-0.

Philipp Bunge. Scripting browsers with glamour. Master's thesis, Bern University, April 2009.

Che06. Checkstyle, December 2006. http://checkstyle.sourceforge.net.

Cincom Systems, Inc. Visualworks smalltalk environment. http://www.cincom.com/smalltalk, 2007.

J. Coplien. *Advance C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

Marc Denecker and Antonis Kakas. Abduction in logic programming. 1993. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.134.7504`.

Peter Flach. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., New York, NY, USA, 1994. ISBN 0-471-94152-2.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Anthony Hunter and Bashar Nuseibeh. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, 1998. URL `citeseer.ist.psu.edu/article/hunter95managing.html`.

Katsumi Inoue and Chiaki Sakama. Specifying transactions for extended abduction. In *In: Proc. 14th Int'l Joint Conf. on Artificial Intelligence*, pages 394–405. Morgan Kaufmann, 1998.

Stephen Johnson. Lint. http://www.jutils.com/, 2007.

John R. Josephson and Susan G. Josephson. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge University Press, 1994.

R.A. Kowalski A.C. Kakas and F. Tony. The role of abduction in logic programming. In C.J. Hogger D.M. gabbay and J.a. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, 1998.

Andy Kellens. *Maintaining causality between design regularities and source code*. PhD thesis, Vrije Universiteit Brussel, 2007.

S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Softw.*, 3(3):41—49, 1986.

Kim Mens and Andy Kellens. IntensiVE, a toolsuite for documenting and testing structural source-code regularities. *10th Conference on Software Maintenance and Re-engineering (CSMR)*, pages 239–248, 2006.

Kim Mens, Andy Kellens, Frederic Pluquet, and Roel Wuyts. The intensional view environment. *International Conference on Software Maintenance (ICSM) Industrial and Tool Volume*, pages 81–84, 2005.

Tim Menzies. Applications of abduction: knowledge-level modelling. *Int. J. Hum.-Comput. Stud.*, 45(3):305–335, 1996. ISSN 1071-5819. doi: http://dx.doi.org/10.1006/ijhc.1996.0054.

Naftaly H. Minsky. Law-governed regularities in object systems: part i: an abstract model. *Theor. Pract. Object Syst.*, 2(4):283–301, 1996. ISSN 1074-3227. doi: http://dx.doi.org/10.1002/(SICI)1096-9942(1996)2:4⟨283::AID-TAPO4⟩3.0.CO;2-V.

Naftaly H. Minsky and H. Minsky. Law-governed regularities in software systems. In *In Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 299–312. Springer-Verlag, 1994.

Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: http://doi.acm.org/10.1145/336512.336523.

Bashar Nuseibeh and Alessandra Russo. Using abduction to evolve inconsistent requirements specifications. In *the Use of Logical Abduction in Software Engineering 25*, pages 1039–7841, 1999.

Bashar Nuseibeh and Alessandra Russo. Using abduction to evolve inconsistent requirements specification. *Australasian Journal of Information Systems*, 6(2), 2007. ISSN 1326-2238. URL `http://journals.sfu.ca/acs/index.php/ajis/article/view/296/266`.

Sami Paavola. Abduction as a logic and methodology of discovery: the importance of strategies. *Foundations of Science*, 9(3):267+, 2004. ISSN 1233-1821. doi: 10.1023/B:FODA.0000042843.48932.25. URL `http://dx.doi.org/10.1023/B:FODA.0000042843.48932.25`.

David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

C. S. Pierce. *The Collected Papers of Charles Sanders Peirce*. Harvard University Press, 1935.

Charles S. Pierce. Abduction and induction. In J. Buchler, editor, *Philosophical Writings of Pierce*, pages 150–156. Dover Books, New York, 1955.

R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, April 1987. ISSN 0004-3702. doi: 10.1016/0004-3702(87)90062-2. URL `http://dx.doi.org/10.1016/0004-3702(87)90062-2`.

M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng*, 30(12):889—903, 2004.

A. Russo and B. Nuseibeh. On the use of logical abduction in software engineering. In S. K. Chang, editor, *Software Engineering and Knowledge Engineering*. World Scientific Publishing Corporation, 2000. URL `citeseer.ist.psu.edu/russo00use.html`.

Alessandra Russo, Rob Miller, Bashar Nuseibeh, and Jeff Kramer. An abductive approach for handling inconsistencies in scr specifications. In *ICSE2000 Workshop on Intelligence Software Engineering*, 2000. URL `http://theory.doc.ic.ac.uk/~ar3/wise3.pdf`.

K. Satoh. Adding and deleting pollution marker by abductive logic programming. In *First Asia Pacific Workshop on Intelligent Software Engineering (APWISE'98)*, pages 48 – 53, 1998a.

K. Satoh. Computing minimal revised logic program by abduction. In *International Workshop on the Principles of Software Evolution, IWPSE98*, pages 177 – 182, 1998b.

K. Satoh. Consistency management in software engineering by abduction. In *ICSE-2000 Workshop on Intelligent Software Engineering*, pages 90 – 99, 2000.

Leon Shapiro and Ehud Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, April 1994. ISBN 0262691639.

P. Sullivan. On falsification interpretation of peirce. 27:197–219, 1991.

J. van Gurp and J. Bosch. Design erosion: Problems & causes. *Systems & Software*, 61(2):105–119, 2001.

Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *In Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

Roel Wuyts and Stephane Ducasse. Unanticipated integration of development tools using the classification model. In *Computer Languages, Systems and Structures 30*, pages 1–2, 2003.

Chong Ho Yu. Abduction? deduction? induction? is there a logic of exploratory data analysis? Technical report, Annual Meeting of American Educational Research Association, April 1994.

Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/237432.237434.

Andrea Zisman and Alexander Kozlenkov. Knowledge base approach to consistency management of uml specifications. In *ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering*, page 359, Washington, DC, USA, 2001. IEEE Computer Society.