Diagnosing and Correcting Design Inconsistencies in Source Code with Logical Abduction

Sergio Castro[1], Coen De Roover[2], Andy Kellens[2], Angela Lozano[1], Kim Mens[1], Theo D'Hondt[2]

Abstract

Correcting design decay in source code is not a trivial task. Diagnosing and subsequently correcting inconsistencies between a software systems's code and its design rules (e.g., database queries are only allowed in the persistence layer) and coding conventions can be complex, time-consuming and error-prone. Providing support for this process is therefore highly desirable, but of a far greater complexity than suggesting basic corrective actions for simplistic implementation problems (like the "declare a local variable for non-declared variable" suggested by Eclipse).

We present an abductive reasoning approach to inconsistency correction that consists of (1) a means for developers to document and verify a system's design and coding rules, (2) an abductive logic reasoner that hypothesizes possible causes of inconsistencies between the system's code and the documented rules and (3) a library of corrective actions for each hypothesized cause. This work builds on our previous work, where we expressed design rules as equality relationships between sets of source code artifacts (e.g., the set of methods in the persistence layer is the same as the set of methods that query the database). In this paper, we generalize our approach to design rules expressed as user-defined binary relationships between two sets of source code artifacts (e.g., every state changing method should invoke a persistence method).

We illustrate our approach on the design of IntensiVE, a tool suite that enables defining sets of source code artifacts intensionally (by means of logic queries) and verifying relationships between such sets.

Keywords: inconsistency management, abductive reasoning, logic meta programming

1. Introduction

When creating, maintaining and evolving software, it is not a trivial task for a developer to ensure that his source code respects the design of the software.

This includes various *design rules* —such as the correct and consistent application of *coding conventions* [4], *idioms* [14], *design patterns* [18] and *design regularities* [33]— that describe how the source code of the system should be structured. Violations of these design rules can cause the software to become harder to understand and maintain, and can sometimes even result in erroneous behaviour. A system using the *Hibernate* persistence framework [3], for example, should adhere to the framework's design rules —at the risk of not functioning properly if those rules are violated.

Testimony to this problem is the wide variety of software tools that support verifying design rules. Tools such as *FindBugs* [12], *CheckStyle* [11] and *Lint* [24] inform developers of violations of common coding guidelines and rules of thumb. Tools like *Ptidej* [19] enable verifying whether design patterns are implemented consistently in the source code. Some tools (e.g., *SmallLint* [6] and *Eclipse's* quick fixes) even suggest corrections for the violations they identify. However, few tools support documenting and verifying user-specified design rules. Examples include

Email addresses: Sergio.Castro@uclouvain.be (Sergio Castro), cderoove@vub.ac.be (Coen De Roover), akellens@vub.ac.be (Andy Kellens), Angela.Lozano@uclouvain.be (Angela Lozano), Kim.Mens@uclouvain.be (Kim Mens), tjdhondt@vub.ac.be (Theo D'Hondt)

[1]Université catholique de Louvain, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium

[2]Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium

IntensiVE [27], *NDepend* [48] and *Semmle* [47]. None of these tools supports correcting violations of user-specified design rules. Typically, a manual effort is required to diagnose and correct the causes of such violations [10].

In this paper we present an approach and its associated tool that supports a three-step process for semi-automatic correction of inconsistencies between *user-specified* design rules and code:

1. Our approach supports documenting a system's design rules and verifying the system's code with respect to the documented rules. To this end, we leverage our earlier work on the IntensiVE tool suite [27] which verifies design rules expressed in the logic meta programming language SOUL [50].
2. Furthermore, the logic meta programming foundations of the IntensiVE tool suite allow diagnosing violations by means of an *abductive logic reasoner* [17]. The abductive logic reasoner hypothesizes possible causes for any violation identified by IntensiVE and gives feedback to the developer.
3. Finally, by associating corrective actions with each of the potential causes reported by the abductive reasoner, our approach enables the (semi-)automated correction of design rule violations.

This work expands on our previous work [9] in which we demonstrated the applicability of abductive logic reasoning for diagnosing and correcting one particular kind of design rule that can be expressed using IntensiVE. The contribution of this paper over the previous work is two-fold. First, we generalize the ideas developed in [9] to support all the different kinds of design rules that can be expressed using IntensiVE. Second, we present a set of dedicated tools that integrate with the IntensiVE tool suite to assist in the diagnosis and correction of inconsistencies between design rules and source code.

This paper is structured as follows. Section 2 provides an overview of the logic meta programming foundations of the IntensiVE tool suite that enable documenting and verifying a system's design rules. Section 3 discusses how any identified inconsistencies between the documented design rules and the source code can be diagnosed using *abductive reasoning*. Section 4 subsequently demonstrates how associating corrective actions with the hypothesized causes of an inconsistency enables the semi-automated correction of design rule violations. Section 5 illustrates the complete diagnosis and correction process supported by our approach using examples taken from the implementation of IntensiVE itself. Section 6 discusses the advantages and disadvantages of our approach. We give an overview of related approaches in Section 7. Before concluding this paper in Section 9, future work is discussed in Section 8.

2. Documenting and Verifying Design Rules using IntensiVE

IntensiVE [8, 27, 30] is a tool suite for documenting structural design rules and verifying their validity with respect to source code. This tool suite has been applied to design rules concerning coding conventions, implementation idioms, design pattern implementations and architectural dependencies.

At the core of IntensiVE lies the concept of an *intensional view*. Such a view is a set of source code artifacts (e.g., classes, methods or fields) that are conceptually related (e.g., all *visit methods* in an instance of the Visitor design pattern). Key to IntensiVE is that this set of artifacts is not defined by explicitly enumerating all of its elements, but rather by means of an intensional description. Concretely, this description is a logic query that, upon evaluation, yields the artifacts belonging to the intensional view. To this end, IntensiVE uses the Smalltalk Open Unification Language (SOUL) [50] which is a Prolog-like logic programming language with specialized features for reasoning about the source code of Smalltalk [7, 31], Java [7, 15], C [16] and COBOL [28] programs.

To document structural design rules, relationships can be expressed between intensional views. IntensiVE supports three kinds of relationships, namely *multiple alternative views*, *unary relationships* and *binary relationships*. In what follows, we take a more in-depth look at each of these kinds of relationships, and illustrate how they can be used to express design rules. We take examples from the implementation of IntensiVE itself, of which the underlying design rules have been documented using intensional views and relationships. In Section 4, we revisit these examples and demonstrate how our approach supports diagnosis and correction of design rule violations in IntensiVE.

2.1. Multiple alternative views

Multiple alternative views are the first kind of relationship. These express equality relationships between multiple intensional views. For each of the alternative views that participate in this kind of relationship, the set of source code artifacts that is obtained by evaluating the definition of that intensional view should be identical. To illustrate

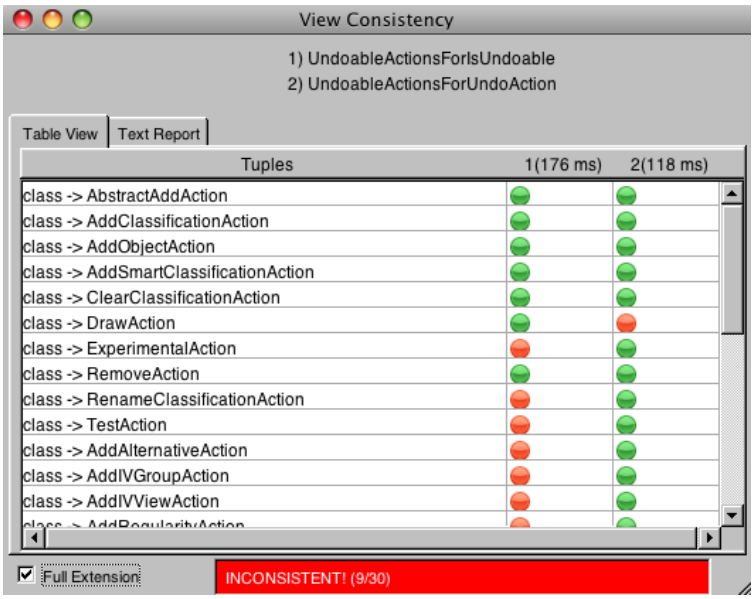

Figure 1: Verification of the *undoable actions* design rule.

this concept, we take a look at the implementation of *undoable actions* in IntensiVE. In order to implement all the user actions within the tool suite, a Command design pattern [18] is used, where each user action is implemented by means of a separate class. A subset of the available actions within IntensiVE are undoable. These undoable actions are characterized by a design rule that states that a class representing an undoable action needs to understand both the messages `isUndoable` and `undoAction`. If a class only understands one of these methods, then the design rule is violated, leading to erratic behaviour of the system. Moreover, the implementation of the `isUndoable` method should consist of a single statement that returns the boolean `true`, in order to mark the action as undoable.

To document this design rule, we introduce two alternative intensional views, defined using the following two SOUL queries:

1. `if classChainUnderstandsMessageWithName(?class,?method,isUndoable), booleanFlagMethod([true],?method)`
2. `if classChainUnderstandsMessageWithName(?class,?,undoAction)`

In solutions to the first query (i.e., the elements of the first alternative intensional view), the logic variable *?class* is bound to a class of which the instances understand message `isUndoable` (i.e. either the class or one of its superclasses implements a method with that name). Moreover, as required by the second condition of the query, method *?method* returns the boolean value `true`. This condition is defined as follows:

```
booleanFlagMethod(?boolean,?method) if
    boolean(?boolean),
    methodWithUniqueStatement(?method,RBReturnNode(RBLiteralValueNode(?boolean)))
```

Predicate `methodWithUniqueStatement/2` is defined as follows:

```
methodWithUniqueStatement(?method,?statement) if
    statementsOfMethod(<?statement>,?method)
```

The `methodWithUniqueStatement/2` predicate verifies that the body of method *?method* consists of a single statement *?statement*.[3]

[3] As SOUL lists are delimited by <>, <*?statement*> is the singleton list with element *?statement*.

Solutions to the query for the second alternative view consist of classes *?class* of which the instances understand the message `undoAction`.

When verifying the validity of the above relationship (i.e., when checking equality of the two resulting sets of classes) with respect to the source code, the IntensiVE tool suite warns its user of all undoable actions that are part of one of the alternative views, but not of the other. Figure 1 shows this kind of feedback. In the left column of the screenshot, all elements belonging to the intensional view are listed. The remaining columns indicate which alternative view a particular artifact belongs to. For instance, the tool reports that the `ExperimentalAction` implements the `undoAction` method, but does not return `true` in the `isUndoable` method.

2.2. *Unary intensional relationships*

Next to multiple alternatives, IntensiVE also supports the notion of unary relationships. Unary relationships are declared over a single intensional view and are used to express a number of conditions that need to be respected by the elements belonging to this intensional view. These relationships are of the form: $Qx \in V : conditions(x)$ where Q is a logic quantifier (e.g., $\forall, \exists, \nexists$), V is an intensional view and *conditions* is logic query expressed using SOUL.

To illustrate the use of unary relationships, we document the design rule that within the source code of IntensiVE, all methods overriding `initialize` should contain a `super` call as the first statement of their implementation, in order to ensure that objects are correctly initialized.

An intensional view named *overridden initialization* is created to document this design rule. It is defined by means of the following SOUL query:

```
if methodWithNameInClass(?method,initialize,?class),
   overridingSelector(?class,initialize)
```

This query will gather all overridden `initialize` methods by first retrieving all the methods *?method* named `initialize` in the system, along with the corresponding class *?class* in which the method is implemented. The second condition restricts these classes to those that actually override the `initialize` method.

The actual design rule is documented by imposing a unary relationship over the intensional view *overridden initialization*. Using the IntensiVE tool suite, this relationship is defined as follows:

```
∀ ?override ∈ Overridden initialization:
        methodBeginsWithSuperCall(?override.method)
```

This unary relationship expresses that for all overrides *?override* that are part of the *Overridden initialization* intensional view, the first statement of the method should be a `super` send. To this end, the condition of this relationship uses predicate `methodBeginsWithSuperCall/1`. This predicate is defined as:

```
methodBeginsWithSuperCall(?method) if
    methodWithName(?method,?name),
    argumentsOfMethod(?args,?method),
    methodWithFirstStatement(?method,
      RBMessageNode(RBVariableNode(super),?name,?args))
```

The third condition verifies if the overridden method *?override.method* has as a first statement a message send (`RBMessageNode`) corresponding to the method name (*?name*) where this message belongs to. The receiver of this message is the pseudo-variable `super` and the arguments list is the same than the received by the host method (*?args*). The predicate `methodWithFirstStatement/2` is defined as follows:

```
methodWithFirstStatement(?method,?statement) if
  statementsOfMethod(<?statement|?>,?method)
```

2.3. *Binary intensional relationships*

Binary intensional relationships are a generalization of the concept of unary relationships. These binary relationships enable expressing design rules that involve two intensional views and that express how the artifacts belonging to one of these views are related to those belonging to the other. Binary relations are of the form:

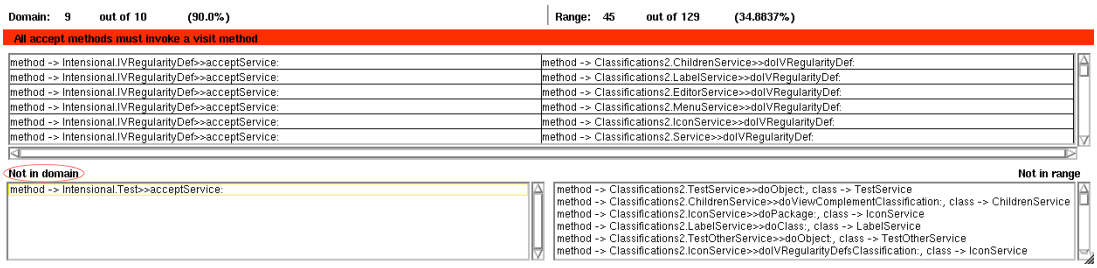

Figure 2: Verification of the binary intensional relation documenting the Visitor design pattern.

$Q_1 x \in V_1; Q_2 y \in V_2 : condition(x, y)$ where Q_1 and Q_2 are logic quantifiers, V_1 and V_2 are intensional views, and `condition` is a SOUL query that expresses a relationship between the elements of the two intensional views.

As an illustration of the use of binary relationships, we discuss the documentation of the instance of the Visitor design pattern [18] that is present in the implementation of IntensiVE. In order for the system to work properly, all the *accept methods* that accept a Visitor should call a corresponding *visit method* on the Visitor using the double dispatching idiom. To document this design rule we rely on the intensional views *Accept methods* and *Visit methods* which are defined as follows.

First, the intensional view *Accept methods* groups all accept methods in the system using the following SOUL query:

```
if methodWithNameInClass(?method,acceptService:, ?)
```

This query gathers all the methods *?method* that are named `acceptService:`. This definition relies on the fact that all accept methods in the system follow this naming convention.

The second intensional view, *Visit methods* is defined using the following query:

```
if classInHierarchyOf(?class,[Service]),
  methodWithNameInClass(?method,{do.*},?class),
```

Visit methods can be characterized as all the methods `?method` that are defined on a class in the hierarchy of `Service`, which is the root class for all the visitors in IntensiVE (the first condition) that start with the prefix 'do'. To this end, the second condition uses the `methodWithNameInClass/3` predicate that checks whether the name of the visit method *?method* defined on class *?class* matches the regular expression `{do.*}`.

```
∀ ?accept.method ∈ Accept methods:
  ∃ ?visit.method ∈ Visit methods:
     methodAcceptsVisitorWithMethod(?accept.method,?visit.method)
```

This relationship expresses that all accept methods *?accept.method* should call a corresponding *?visit.method* using a double dispatch.

The predicate `methodAcceptsVisitorWithMethod/2` is defined with:

```
methodAcceptsVisitorWithMethod(?acceptMethod,?visitMethod) if
    argumentsOfMethod(<?argument>,?acceptMethod),
    methodWithName(?visitMethod,?visitMethodName),
    methodWithUniqueStatement(?acceptMethod,
      RBReturnNode(RBMessageNode(?argument,?visitMethodName,<RBVariableNode(self)>)))
```

This is verified by a SOUL query that comprises three conditions. The first condition checks whether the *?accept.method* takes a single argument *?argument*. In the second condition, the name of the *?visit.method* is retrieved and bound to the logic variable *?visitMethodName*. Finally, the third condition verifies if the *?accept.method* consists of a single statement that returns the result of sending the message *?visitMethodName* to the first *?argument* of the accept

method, with as argument the pseudo-variable `self`. Note that this last condition expresses the double dispatching idiom.

Similar to the verification of multiple alternatives, IntensiVE provides a dedicated sub-tool for verifying unary and binary relationships. Figure 2 depicts the feedback provided by this tool. The top of the pane lists the pairs of elements from both intensional views for which the relation holds. Underneath this list, there are two boxes named *not in domain* and *not in range* that respectively contain the elements belonging to either of the intensional views, but that do not participate in the intensional relation. In the screenshot, the list of artifacts in the *not in domain* box contains all the accept methods that do not invoke a corresponding visit method using a double dispatch.

3. Diagnosing Violations of Design Rules using Abductive Reasoning

As discussed in the previous section, the IntensiVE tool suite serves as our means to document a system's design rules and subsequently verify their validity with respect to its source code. This section introduces abductive logic reasoning as a means to hypothesize possible causes of inconsistencies between the documented rules and the source code. The third component of our approach, a library of corrective actions for each hypothesized cause, will be discussed in the next section.

3.1. Abductive Reasoning

Abductive reasoning was formally introduced by Pierce as one of the fundamental forms of human reasoning (the others being *deduction* and *induction*) [38]. Intuitively, it can be defined as the scientific discovery of explanatory hypotheses for anomalous phenomena or observations [37, 39]. In other words, abductive reasoning is suited for finding a set of hypotheses (i.e. an explanation) that, when added to a given logic theory, would allow an observation to be inferred [43].

Formally, an *abductive logic theory* can be defined as a tuple (P, A) where P is a logic program and A is a set of atoms referred to as *abducibles*. The set of ground atoms $\Delta \subseteq A$ is an hypothesis for the observation Q if:

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

In other words, an abductive explanation Δ together with the original theory P should satisfy the observation Q we want to explain. Note that abductive reasoning only produces *ground* atoms as hypotheses.

An example adapted from [17] and [23] illustrates abductive reasoning. Consider one would like to explain why an object is flying (i.e., the observation). A possible hypothesis explaining this observation is that the object must be bird. Together with the logic theory that a bird flies, this hypothesis explains the observation. However, other hypotheses are possible. For instance, the object could be an airplane. Therefore, the hypothesis that the object must be bird is *defeasible* (i.e., its validity can be refuted when further knowledge is acquired). Strategies can be devised to guide the selection of appropriate hypotheses [37]. For instance, the hypothesis about the object being an airplane can be discarded if the flying object is known to have feathers.

Traditional abduction only hypothesizes ground atoms that will explain an observation once they are added to the logic theory [40]. Extended abduction [23] generalizes abduction to explain negative observations. Moreover, it introduces negative hypotheses (i.e., ground atoms that will explain an observation once they are retracted from the logic theory).

3.2. Abductive Resolution Procedure

The following abductive logic theory (P, A) corresponds to our example of an object that is observed to be flying:

- the logic program P:

```
flies(?x) if bird(?x)
```

- the set of abducibles A:

```
{bird(?x)}
```

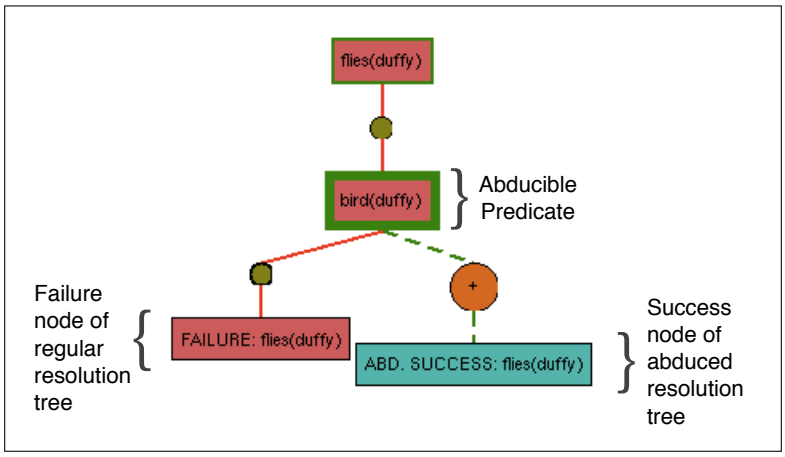

Figure 3: Visualization of the resolution process with abduction.

Figure 3 illustrates how an abductive explanation Δ: {`bird(duffy)`} for the observation Q: `flies(duffy)` can be found procedurally using an abductive extension of the regular resolution procedure for Prolog programs. Each rectangle in Figure 3 represents a different stage of the resolution process. Informally, the abductive resolution procedure continues where the regular resolution procedure fails by adding the failed goal to the explanation Δ for the observation Q —if the failed goal is an abducible atom.[4] Using the regular resolution procedure, the goal `flies(duffy)` fails because its sub-goal `bird(duffy)` fails. In the figure, this is indicated with an uninterrupted edge between node `bird(duffy)` and the failure node in the bottom left corner. Rather than failing at node `bird(duffy)`, the abductive resolution procedure continues by adding `bird(duffy)` as an explanation for the observation Q. This is indicated with an interrupted edge between the node and the abductive success node in the bottom right corner.

Figure 4 illustrates how an explanation Δ: {`-bird(tweety)`} (i.e. one that exists of a negative hypothesis that will explain the observation once it is retracted from the logic theory) for the observation Q: `flies(tweety)` can be found procedurally using a resolution procedure for extended abduction —with the abductive logic programming theory (P, A) modified as follows:

- the logic program P:

```
flies(?x) if bird(?x), not(abnormal(?x))
abnormal(?x) if hasBrokenWing(?x)
bird(tweety)
hasBrokenWing(tweety)
```

- the set of abducibles A:

```
{bird(?x), hasBrokenWing(?x)}
```

Using the regular resolution procedure, the goal `flies(tweety)` fails as its sub-goal `not(abnormal(tweety))` fails because `hasBrokenWing(tweety)` succeeds. In the figure, this is indicated with an uninterrupted edge between node `not(abnormal(tweety))` and the failure node in the bottom left corner. Rather than failing at node `not(abnormal(tweety))`, the abductive resolution procedure succeeds by retracting `hasBrokenWing(tweety)` from the theory. This is possible because `hasBrokenWing/1`[5] has been declared as an abducible. Note that circles in the figure denote transitions between notes (i.e., the clause that was used to pass from one node to another in the resolution process). Transitions to an abduced node are larger than the other transitions —containing either a + or a - sign denoting that a ground atom was added to the theory or retracted from the theory respectively.

[4] We refer to [17] for a complete description of the abductive resolution procedure.

[5] Throughout this paper we use the notation `predicate/arity` where `predicate` represents the name of the predicate and `arity` the number of arguments of the predicate.

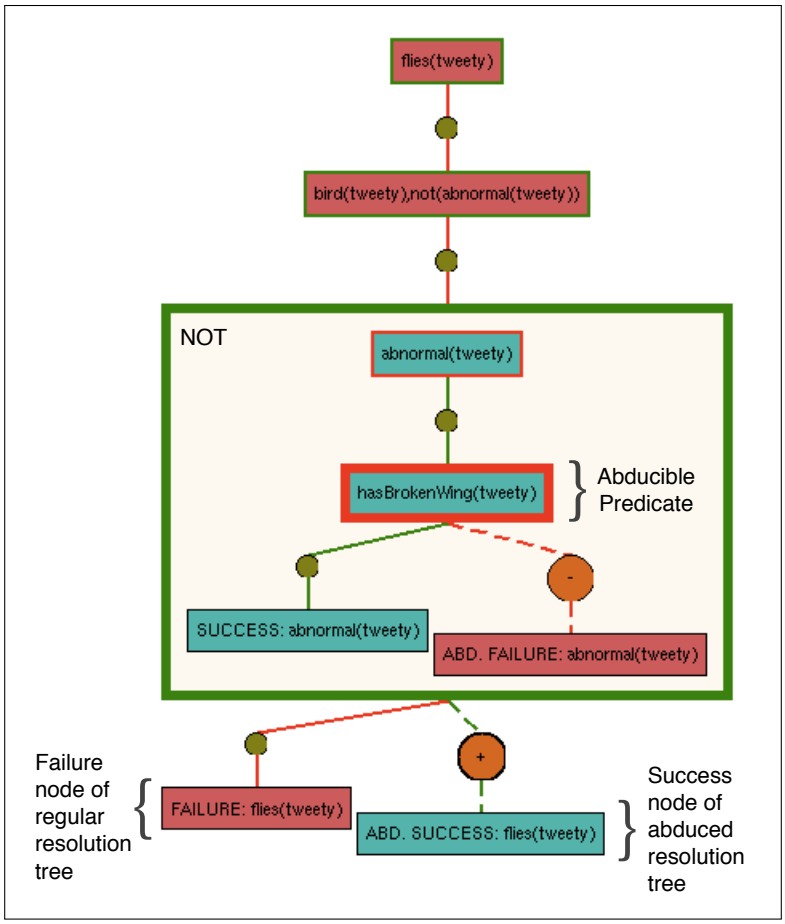

Figure 4: Visualization of the resolution process with extended abduction.

3.3. Diagnosing Violations of Design Rules

Reiter [41] defines diagnosis as the search of the components of a system that —under the assumption that these components are faulty— will explain inconsistencies between a description of this system and certain observations. Following Reiter's definition, we assume that violations of design rules are caused by faults in the system's components. We do not consider the possibility that the actual documentation of these design rules is incorrect.

The IntensiVE tool suite discussed in Section 2 documents design rules as relations between sets of source code elements. The logic meta programming foundation of this suite (i.e., it relies on logic queries that quantify over the program's code to define those sets and relations) allows diagnosing design rule violations by means of abductive logic reasoning. IntensiVE reports design rule violations as a set of source code elements for which the relations do not hold. To diagnose a violation, it therefore suffices to re-evaluate the logic queries that define the violated relations —this time using abductive logic resolution with the free variables of the query bound to the source code elements that correspond to the violation.

To illustrate how this works in practice, consider the inconsistencies of the *undoable actions* design rule (cf. Section 2.1) reported by IntensiVE (depicted in Figure 1). In particular, we take a more detailed look at the `ExperimentalAction` class. It is reported as violating the design rule because it implements method (it is an element of the second alternative intensional view), but it does not answer message `isUndoable` with the boolean `true` (it is not contained in the first alternative intensional view). To diagnose this violation, the logic queries defining the alternative intensional views have to be re-evaluated with logic variable *?class* bound to class `ExperimentalAction` —this time using abductive rather than regular resolution.

Assuming predicates `statementsOfMethod/2` and `methodWithNameInClass/3` are declared abducibles, the abductive logic interpreter diagnoses this violation with the explanations:

- Δ_1:

```
{+statementsOfMethod(<RBReturnNode(RBLiteralValueNode([true]))>,
                     [ExperimentalAction>>isUndoable])}
```

- Δ_2:

```
{-methodWithNameInClass([ExperimentalAction>>undoAction],
                        undoAction,
                        [ExperimentalAction])}
```

In other words, the interpreter has hypothesized two possible causes for the `ExperimentalAction` violation: the fact that its `isUndoable` method does not return the boolean `true`, or the fact that the class defines an `undoAction` method. Hence, class `ExperimentalAction` should be altered to comply with the design rule by either changing its implementation of the `isUndoable` method or deleting the method `undoAction`. The final decision is left to the programmer, but previously defined corrective actions associated with these abducible predicattes simplify this task.

3.3.1. Declaring Reification Predicates as Abducibles

SOUL offers several predicate libraries for reasoning about a system's source code. Each library provides predicates that reify the basic relations between the nodes in an abstract syntax tree representation of the program (e.g., `methodWithNameInClass/3`). In addition, each library provides higher-level predicates that quantify over relations between AST nodes that are not explicit in the AST representation. Examples include `classChainUnderstandsMessageWithName/2` and `methodBeginsWithSuperCall/1`. As the higher-level predicates are implemented in terms of the reification predicates, the abductive logic reasoner only considers the latter as abducible predicates —each explanation should be minimal. Implementation-wise, predicates are declared abducible using rules of the following form:

```
abducible(?abductiveIntention(?abduciblePredicate)) if ?groundingConditions
```

where *?abductiveIntention* is the constant `in` or `out` indicating whether *?abduciblePredicate* has to be added to or removed from the logic theory respectively. Although *?abduciblePredicate* can contain variables, explanations comprise

ground atoms only. Logic variables within *?abducible Predicate* are therefore either bound in the query evaluated by the abductive logic interpreter (i.e., the one corresponding to a design rule violation) or bound via the *?groundingConditions* conditions. For instance, `true` and `false` are the possible bindings for a variable corresponding to a boolean value in abducible predicate `methodReturnsBoolean/2`:

```
abducible(in(methodReturnsBoolean(?method, ?boolean)) if
    nonVar(?method),
    method(?method),
    boolean(?boolean)

boolean(true)
boolean(false)
```

4. Corrective Actions for Violations of Design Rules

In the previous section, we discussed how abductive logic reasoning enables diagnosing violations of design rules documented by means of relations between intensionally defined (i.e., by means of a logic query) sets of source code elements. This section describes how associating corrective actions (i.e., source code transformations) with each hypothesis enables the semi-automatic correction of inconsistencies between a systems's design and its code. In our library, corrective actions are declared as rules of the following form:

```
correct(?abducedHypothesis, ?action) if ?generatorConditions
```

Here, *?abducedHypothesis* is an hypothesis from an abduced explanation (a ground atom corresponding to an abducible predicate) and *?action* a Smalltalk expression performing the corrective action upon the user's request. An abduced explanation consists of abducible predicates that will explain an observation once they are added to, or removed from the logic theory. In our problem setting, those abducibles are ground reification predicates that reify the relations between the nodes in an abstract syntax tree representation of the program. An explanation that requires a predicate to be added to or removed from the logic theory implies changes to the program's AST. Rather than changing the reification of the program and subsequently constructing the modified program from the updated reification, we directly change the program's AST itself. This is possible because SOUL's symbiosis with Smalltalk enables reifying an AST node as the AST node itself (i.e., a Smalltalk object rather than a compound term) [7]. Hence, most corrective actions can be performed by sending messages to AST nodes. To preserve the natural use of unification when quantifying over AST nodes, a reified AST node unifies with a structurally equivalent compound term [7].

Note that multiple corrective actions can be associated with a single abduced hypothesis. For a particular hypothesis *?abducedHypothesis* selected by a user, our tool presents all bindings for variable *?action* in solutions to the query:

```
if correct(?abducedHypothesis, ?action)
```

In the remainder of this section, three corrective actions will be defined. They will be used afterwards in the examples presented in Section 5.

4.1. Corrective Action for `statementsOfMethod`

The corrective action associated with an abduced hypothesis `statementsOfMethod/2` is declared as follows:

```
correct(in(statementsOfMethod(?modelStatements, ?method), ?action) if
  methodSignature(?method,?signature),
  statementsOfMethod(?oldStatements,?method),
  unifyStatements(?oldStatements,?modelStatements,?newStatements),
  methodSourceCode(?signature,?newStatements,?newMethodCode),
  equals(?action, [[?class compile: ?newMethodCode]])
```

This corrective action executes a code transformation that will change the statements of method *?method* according to the statements *?modelStatements*. The first two conditions gather information about the current state of the method: its signature and its statements. Note that *?modelStatements* could contain free variables. In that case, these variables should be unified with the original values in the original statements of the method. Otherwise, original values will be replaced by the values in *?modelStatements*. This unification is performed by the third condition: `unifyStatements/3`. The new code for the method is found in the fourth condition. The last condition creates a corrective action. This is a Smalltalk block that —when requested by the user— instructs class *?class* to recompile the source code of a method, thus correcting the inconsistency.

4.2. Corrective Action for `methodWithNameInClass`

The corrective action associated with an abduced hypothesis `methodWithNameInClass/3` is declared as follows:

```
correct(out(methodWithNameInClass(?method, ?name, ?class), ?action) if
      methodWithNameInClass(?method, ?name, ?class),
      equals(?action, [[MLI forSmalltalk removeMethod: ?method]])
```

The corrective action *?action* removes method *?method* from the systems's code. Upon execution of this corrective action, its associated abducible predicate will be retracted from the theory. Note that variable *?method* will be bound by the first condition if necessary.

4.3. Corrective Action for `methodInProtocol`

The corrective action associated with an abduced hypothesis `methodInProtocol/2` is declared as follows:

```
correct(in(methodInProtocol(?method, ?protocol), ?action) if
      methodWithNameInClass(?method, ?, ?class),
      methodSourceCode(?method, ?methodCode),
      equals(?action, [[MLI forSmalltalk
          compileMethodInClass: ?class inProtocol: ?protocol withCode: ?methodCode]])
```

This corrective action moves method *?method* to protocol *?protocol* by instructing *?class* to recompile the method in the protocol *?protocol*. Note that *?class* will be bound in the first condition.

5. Illustrative Examples

In this section, we illustrate the complete diagnosis and correction process supported by our approach on examples taken from the implementation of IntensiVE itself. The first three examples in this section are structured according to the three kinds of relationships that are supported by IntensiVE for documenting design rules: multiple alternative views, unary relations and binary relations. For each of these kinds of relationships, we revisit an example from Section 2 and demonstrate how our approach diagnoses the problem and proposes corrective actions. Afterwards, we take a look at a more complex example that illustrates how our approach supports corrective actions that consist of multiple steps, and that intervene at multiple locations in the source code.

5.1. Correcting Inconsistencies in Multiple Alternative Views: Undoable Actions

We start by revisiting the *undoable actions* design rule which states that a class representing an undoable action needs to understand both the messages `isUndoable` and `undoAction`. In Section 2.1, we documented this design rule by means of multiple alternative views. The violations reported by IntensiVE for this design rule are depicted in Figure 1. Class `ExperimentalAction`, for instance, was reported as a violation because it implements method `undoAction`, but does not answer message `isUndoable` with the boolean `true`. In Section 3.3, we discussed how the abductive logic interpreter diagnoses the violation of the `ExperimentalAction` class with explanations

- Δ_1:

```
{+statementsOfMethod(<RBReturnNode(RBLiteralValueNode([true]))>,
                     [ExperimentalAction>>isUndoable])}
```

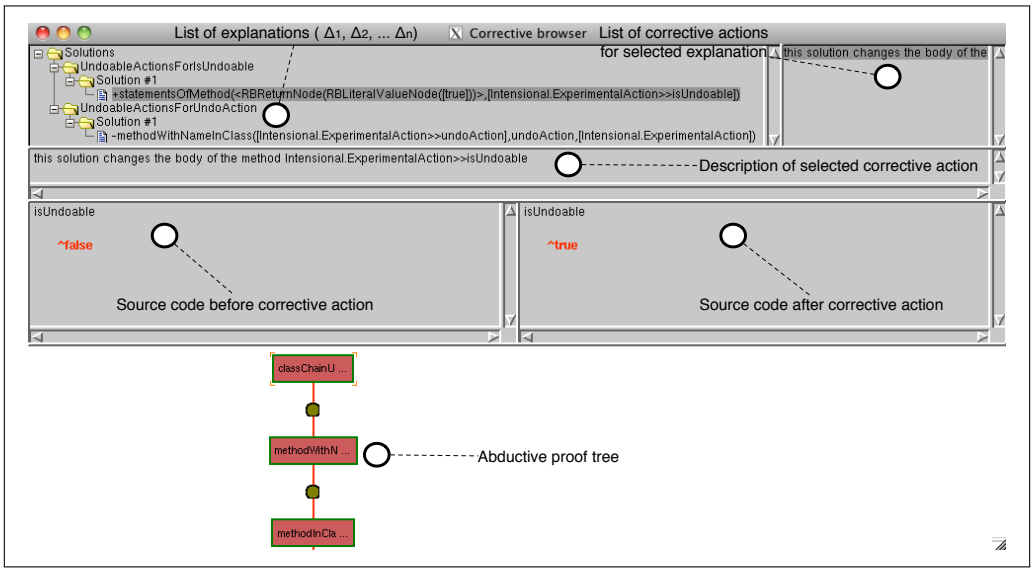

Figure 5: Correcting the `ExperimentalAction` violation of the *undoable actions* design rule.

—meaning that this class can be made to comply with the design rule by ensuring method `isUndoable` returns `true`.

- Δ_2:

```
{-methodWithNameInClass([ExperimentalAction>>undoAction],
                        undoAction,
                        [ExperimentalAction])}
```

—meaning that this class can be made to comply with the design rule by deleting the method `undoAction`.

The above explanations reflect the two possible solutions to the design rule violation. The corrective action associated with explanation Δ_1 will result in class `ExperimentalAction` ending up in both alternative views rather than only in the second. To find explanation Δ_1, the abductive reasoner evaluated the query defining the first alternative view with the violating class substituting for variable *?class*:

```
if classChainUnderstandsMessageWithName([ExperimentalAction],?method, isUndoable),
   booleanFlagMethod([true], ?method)
```

The corrective action associated with explanation Δ_2 will result in class `ExperimentalAction` being removed from the second alternative view —hence no longer being present in any of the alternatives. To find this explanation, the abductive reasoner evaluated the negation of the query defining the second alternative view with the violating class substituting for variable *?class*:

```
if not(classChainUnderstandsMessageWithName([ExperimentalAction], ?, undoAction))
```

Figure 5 depicts the actual feedback provided by our tool. The bottom pane contains the top half of the abductive proof tree for the first explanation. It is drawn following the conventions introduced in Section 3.

The pane in the top-left corner lists the abduced explanations for the `ExperimentalAction` violation. Note that the two explanations are shown: Δ_1 which adds `ExperimentalAction` to the alternative containing all classes that understand message `isUndoable` and return `true` (alternative `UndoableActionsForIsUndoable`) and Δ_2 which removes `ExperimentalAction` from the alternative containing all classes understanding message `undoAction` (alternative `UndoableActionsForUndoAction`).

Once users have selected an explanation, they are offered the corresponding list of corrective actions in the next pane. These actions are gathered by evaluating the query:

```
if correct(statementsOfMethod(<RBReturnNode(RBLiteralValueNode([true]))>,
          [ExperimentalAction>>isUndoable]),
          ?action)
```

In the screenshot, the user has selected the first explanation which will transform the implementation of method `isUndoable`. The second row displays a textual description of the effects of executing the currently selected corrective action. The third row of the tool displays the violating source code element before (on the left) and after (on the right) the corrective action.

5.2. Correcting Inconsistencies in Unary Relationships: Overridden Initialization

By means of a unary relationship imposed on top of the *overridden initialization* intensional view, the design rule stating that methods overriding `initialize` should contain a `super` call as their first statement was documented in Section 2.2. IntensiVE identified method `IntensionalViewLayoutManager>>initialize` as one of the violations of this design rule. The abductive logic interpreter diagnosed this particular violation with the following singleton set as explanation Δ:

```
statementsOfMethod(<RBMessageNode(RBVariableNode(super),initialize,<>)|?>,
                   [IntensionalViewLayoutManager>>initialize])
```

This explanation indicates that the method `IntensionalViewLayoutManager>>initialize` can be made to comply with the design rule by ensuring that its first statement is a super call to initialize (i.e., an `RBMessageNode`). To this end, the abductive logic interpreter evaluated the query part of the unary relationship defined in Section 2.2 with `IntensionalViewLayoutManager>>initialize`: substituting for *?override.method*:

```
if methodBeginsWithSuperCall([IntensionalViewLayoutManager>>initialize])
```

The interpreter was therefore able to abduce the explanation for this violation as predicate `statementsOfMethod/2` is declared as an abducible in our library of reification predicates. Figure 6 depicts the actual feedback provided by our tool for the `IntensionalViewLayoutManager>>initialize` violation. As before, the second pane at the top lists the corrective actions for the abduced explanation that is selected in the first pane. At the second row, we can see in the first column the faulty implementation of the method (without a super call) and in the second column the method with the correct code, having as a first statement a super call.

5.3. Correcting Inconsistencies in Binary Relationships: Accept Methods and Visit Methods

By means of a binary relationship imposed between the *accept methods* and *visit methods* intensional views, Section 2.3 documented the design rule that states that all *accept methods* accepting a Visitor should call a corresponding *visit method* on the Visitor using the double dispatching idiom. Figure 2 depicts the violations of this design rule as identified by IntensiVE. Method `Test>>acceptService:` was, for instance, identified as a violation. It belongs to the *accept methods* intensional view, but does not invoke a visit method using double dispatching (i.e., it is not in the domain of the binary relation).

To diagnose this violation, the abductive logic interpreter had to re-evaluate the logic query that defines the binary relationship between the two views —this time with bindings for variables *?visit.method* and *?accept.method* that substitute for an element from the relationship's domain and an element from the relationship's range respectively. Therefore, variable *?visit.method* should be bound to `Test>>acceptService:`. However, it is unclear what binding should be provided for the variable *?accept.method*. We therefore offer users a list of all the elements in the relationship's range such that they can select the element that corresponds to the violating element in the relationship's domain —assuming that the corresponding method is indeed available in the relationship's range. We do not yet address the situation in which a violation of a binary relationship between two views can only be corrected by adding a new element to the view in the range of the relation. As we selected method `TestService>>doObject:` to complete the binary relation, the abductive logic interpreter diagnosed the `Test>>acceptService:` violation of the relation by evaluating the following query:

```
if methodAcceptsVisitorWithMethod([Test>>acceptService:],[TestService>>doObject:])
```

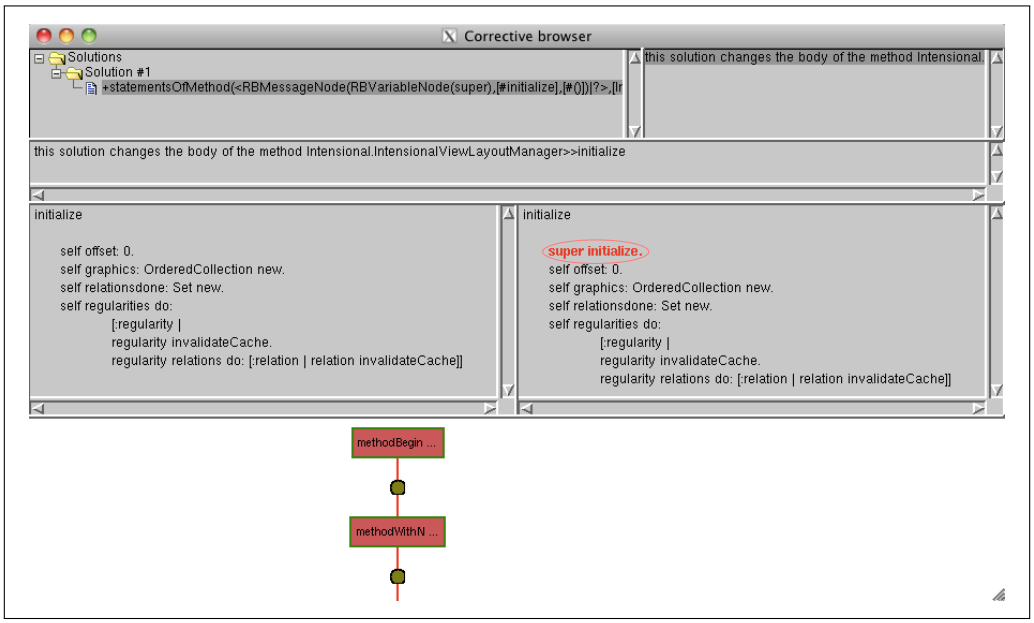

Figure 6: Correcting the `IntensionalViewLayoutManager>>initialize` violation of the *overridden initialization* design rule.

Its explanation for this design violation is the following singleton set Δ:

```
statementsOfMethod(<RBReturnNode(RBMessageNode([RBVariableNode(anObject)],
                                                [#doObject:],
                                                <RBVariableNode(self)>))>,
                   [Test>>acceptService:])
```

Figure 7 depicts the actual feedback provided by our tool. The corrective action associated with the selected hypothesis will alter method `Test>>acceptService` such that its only statement is the return statement (i.e., a `RBReturnNode`) that initiates the double dispatching protocol —as required by the design rule that was violated.

5.4. More complex corrective actions: IntensiVE's saving mechanism

IntensiVE offers a mechanism to store documented regularities on persistent storage. This saving mechanism is implemented by each of the various entities in IntensiVE. Entities have to implement a method `saveOn:` that —by means of a double dispatch—has to invoke *a* method in the `saving mechanism` protocol of the class `IntensionalRootProject`. The prototypical implementation of the saving mechanism is documented using a unary relationship that is defined as:

```
∀ ?save.method ∈ Save methods:
  argumentsOfMethod(<?arg>,?save.method),
  methodWithUniqueStatement(?save.method,RBMessageNode(?arg,
                                                        ?message,
                                                        <RBVariableNode(self)>)),
  methodWithNameInClass(?saving,?message,[IntensionalRootProject]),
  methodInProtocol(?saving,[#'saving mechanism'])
```

The first two conditions of the definition of this unary relation express the double dispatch protocol: they require that the method contains as a unique statement a message send to the single argument of the save method, with `self` as an argument. The other two conditions verify that the message invoked by the double dispatch is implemented in the protocol `saving mechanism` of class `IntensionalRootProject`.

Verification of the above relation revealed that method `saveIn:` of class `RootView` violated the design regularity. A manual inspection of this method indicated two problems:

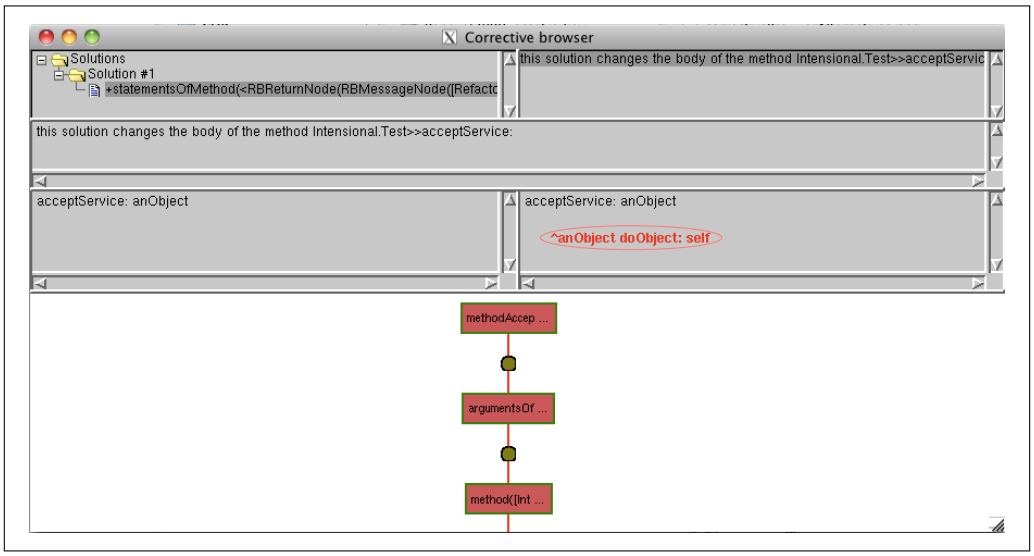

Figure 7: Correcting an inconsistency in the *visitor* pattern regularity.

1. The method violated the double dispatch protocol by passing another object than `self` as argument of the call;
2. The message that was invoked on `IntensionalRootProject` was not classified in the protocol `saving mechanism`.

After abducing an explanation for the reported inconsistency, our tool presented the user with the following possible sequence of corrective actions:

1. Change the argument of the double dispatch to `self`;
2. Move the invoked method on `IntensionalRootProject` to the protocol `saving mechanism`.

Its explanation for this design violation is the following set Δ:

```
statementsOfMethod(<RBMessageNode([RBVariableNode(aLayer)],
                                  [#saveBUG:],
                                  <RBVariableNode(self)>)>,
                   [RootView>>saveIn:]),
methodInProtocol([IntensionalRootProject>>saveBUG:],[#'saving mechanism'])
```

Figure 8 illustrates how our tool reports this explanation. Note that this example illustrates the correction of a more complex design rule. In contrast to the examples above, which involved a single action to be taken to correct a violation, correcting this violation of the saving mechanism design rule required two corrective actions. Moreover, these corrective actions intervene at two separate locations in the source code: method `IntensionalRootProject>>saveBUG:` and method `RootView>>saveIn:`.

6. Discussion

Scalability. We begin our discussion with some scalability issues that might arise and possible solutions. The use of an abductive logic reasoning engine is inherent to our approach. Similar to a regular logic reasoner, an abductive reasoner constructs a proof tree for each of the solutions to a logic query. However, when a branch in the proof tree leads to a failure, the abductive reasoner will hypothesize —based on the declared abducibles— possible changes to the logic theory that would make the query succeed. In our setting, a large search space might therefore need to be explored in order to abduce an explanation for a violation of a design rule. This might hamper the scalability of our approach when analyzing violations of complex design rules. In practice, however, we found that our specific use of abduction limits the search space that needs to be explored. As they are launched against the violating software entity,

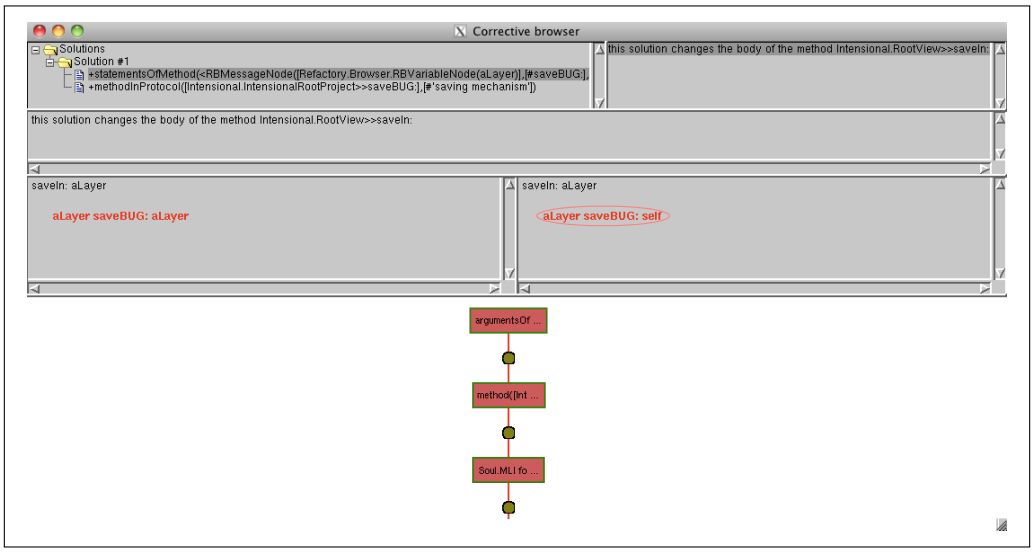

Figure 8: Correcting an inconsistency with multiple corrective actions in the *saving mechanism* regularity.

most queries that serve as input to the abductive reasoner contain only few unbound variables. Nevertheless, our current implementation of the abductive reasoner is suboptimal as it is based on a meta-interpreter on top of SOUL. Consequently, a native implementation of an abductive logic reasoner would provide a significant speed-up.

Another scalability issue concerns the amount of information that is presented to the user. Presenting many corrective actions for a single violation might be overwhelming. We will revisit this issue in the discussion of our future work.

A semi-automated approach. Our approach to diagnosing and correcting violations of design rules cannot be fully automated. First of all, most of our corrective actions are not behaviour-preserving. Developers therefore need to retain control over the transformations that are applied to the source code. Moreover, the abductive reasoner usually proposes multiple sequences of corrective actions for the same violation. While we envision ranking or filtering these corrections based on heuristics, automatically selecting and applying an optimal sequence of corrective actions does not seem feasible. Finally, particular corrective actions require further input from the user (e.g., specifying the name of a method, selecting a corresponding element for a binary relation).

Correctness of the proposed corrective actions. Our approach can propose corrective actions that, although they correct the hypothesized cause of a design rule violation, introduce *other* violations in the source code. This is a well-known limitation of abductive logic reasoning [20] that is caused by the fact that the abductive process is local to the failure branch in the original proof tree. We revisit this issue in the future work section of this paper.

Library of abducibles and corrective actions. As mentioned above, SOUL comes bundled with several predicate libraries for reasoning about Smalltalk, Java, C(++) and COBOL programs. Our tool currently declares the predicates that reify Smalltalk programs as abducibles, and associates generic corrective actions with each abducible. Defining additional abducibles and corrective actions comprises a considerable, but one-time effort. Once abducibles and corrective actions have been provided for a library of predicates, our approach is able to diagnose and correct any violations of design rules that rely on those predicates or on other predicates that use them. Consequently, this is all that would be required to port our approach to the other languages supported by SOUL.

In this paper, we have only considered SOUL's reification predicates (i.e., those that reify AST nodes) as abducibles. The corresponding corrective actions are of an equally fine-grained nature. Violations of rules that rely on higher-level predicates are therefore corrected using a sequence of fine-grained actions —each corresponding to a reification predicate used by the higher-level ones. Alternatively, a higher-level predicate could be declared as an abducible and a coarse-grained corresponding corrective action (i.e., one that manipulates several AST nodes at once) could be provided. Such an effort might be worthwhile if the higher-level predicate is used frequently.

7. Related Work

7.1. Detecting and Correcting Violations of Design Rules

As mentioned in the introduction, there are many tools that verify common programming best practices and rules of thumb in source code [10] (e.g., *FindBugs* [12], *CheckStyle* [11] and *Lint* [24]). Some tools even suggest corrections to well-known problems when they are detected (e.g., *SmallLint* [6] and *Eclipse's* quick fixes). However, all of these tools are limited to a fixed set of problems that can be detected and corrected.

Code querying tools (e.g., *NDepend* [48] and *Semmle* [47]) are ideally suited for documenting custom design rules. Typically, however, developers have to detect violations of a design rule by manually reviewing the results of the corresponding program queries. Architectural and design conformance checkers provide a means to verify a high-level description of a software system (i.e., design patterns, architectural patterns, ...) with the implementation. Examples of such approaches are Reflexion Models [34] and Save Life [29]. To the best of our knowledge, none of these tools offer any semi-automatic support for correcting design violations.

Ptidej [19] is able to detect and subsequently correct imperfect instances of design patterns. To this end, its detection mechanism automatically relaxes constraints from the pattern's specification. Constraints that had to be relaxed in order to find an imperfect pattern instance are considered symptomatic of design defects. *Ptidej* therefore associates a JavaXL [1] transformation rule with each constraint. By applying the transformation rules associated with the constraints that had to be relaxed, imperfect pattern instances can be corrected. *Ptidej*, however, only supports a predefined set of design patterns and does not consider the program's source code at the level of individual statements.

7.2. Abductive Reasoning in Software Engineering

To the best of our knowledge, abductive logic reasoning has not yet been applied to the diagnosis and semi-automatic correction of inconsistencies between a system's design and its source code.

Russo et al. [43], present a comprehensive survey on applications of abductive logic reasoning in software engineering. Most of the discussed works focus on inconsistency management in requirements engineering [35], particularly the analysis of specifications (i.e., consistency analysis of the model of a system) and the revision of such specifications (i.e., impact of changes in an initially consistent specification) [32, 36, 42, 44, 45, 46].

In general, analysis of the consistency of a specification is accomplished by verifying whether certain properties of a system hold over a model of the system [52]. Abduction can be used to verify, for any property $P(X)$, if the goal $P(X) \wedge \neg P(X)$ can be inferred from a specification. In that case, an inconsistency exists. The explanation of this goal constitutes a diagnosis of why the specification is inconsistent.

Alternatively, in [42] inconsistencies are detected and diagnosed by 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.

In the same survey, inconsistency management techniques using abductive reasoning are divided according to the consistency assumptions they make about the model to examine. For example, the work in [36] describes a specification as a composition of multiple partial specifications with or without logical inconsistencies. 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 [21] is used as a mechanism for reasoning in a inconsistency system without *trivialization* (i.e., without inferring arbitrary information from an inconsistent specification) [5].

Regarding revision of specifications [45, 46, 44], this activity is responsible for re-establishing consistency in order to accommodate a given change request. Abduction in this case is used to identify additional changes on a given specification that should re-establish consistency. In [44] Satoh describes a logic approach based on abductive reasoning for adding and deleting *pollution markers* [2] from a given specification in order to manage consistency after a change has been performed. The objective of these pollution markers is considering inconsistencies as exceptions that can be isolated from the rest of the data.

Finally, abduction has not only been demonstrated a useful technique for detecting and diagnosing inconsistencies, but also as a mechanism for correcting them. Using abduction for *theory change* [22], it is possible to identify consistent changes to be performed on a theory so that a change request is satisfied. This has been applied in the domain of databases. Particularly, abductive reasoning has been shown to be suitable for solving the problem of updating a deductive database [25] (a generalization of the view update problem in relational databases, which has also been solved using abduction in [13]).

In the domain of software modeling, Zisman et al. discuss in [53] a mechanism for checking inconsistencies between UML specifications. Mapping related UML specifications in XMI format, the work uses abduction for declaring as consistency properties certain goals that should succeed following a particular course of events. If a goal does not succeed or if its proof follows another course of events than the one expected, a change-set is abduced which indicates what axioms are to be deleted or added.

8. Future work

Our future work will focus on improving the usability of our tool. Particularly, we will investigate techniques that aid in selecting the "best" sequence of corrective actions from those that are proposed by the abductive reasoner. Moreover, we will investigate techniques to prune undesirable corrective actions during the abduction process.

Ranking corrective actions. While our approach is able to hypothesize corrective actions for violations of a design rule, it does not aid in deciding which of these corrective actions to adopt. Traditionally, this well-known limitation of abductive logic reasoning is mitigated using heuristic strategies [49, 51, 37].

In our problem setting, we envision the following particular strategy. When verifying a design rule, IntensiVE does not only report its violations, but also provides the source code entities that do satisfy the rule. While not taken into account by the abduction process, these complying entities can be used to rank the actions that correct the violating entity. Corrective actions that would render the violating entity more similar to the complying entities, for instance, could be ranked higher.

Pruning corrective actions. As mentioned above, performing an action intended to correct one particular violation can result in another design rule being violated. As a means to circumvent this problem, Kakas [26] proposes to specify additional integrity constraints that verify whether abduced hypotheses do not result in new violations. Such integrity constraints preclude actions that would result in additional violations from being proposed.

Given the context of our work, all of the design rules that are documented using the IntensiVE tool suite can be considered as additional integrity constraints. Furthermore, the unit tests of a system can be incorporated as integrity constraints in a similar way. Corrective actions that result in a failing test to succeed could be favored. Corrective actions that result in failure of tests could be pruned.

9. Conclusion

In this paper we have presented an approach based on abductive logic reasoning for diagnosing and correcting violations of design rules. Design rules are documented as relationships between sets of source code elements. Key to our approach is that such sets are defined intensionally through a logic query that quantifies over the program's source code. By means of logic abduction over the logic query that defines a design rule, our approach hypotheses possible causes for each violation of a design rule. In order to correct the hypothesized causes of an inconsistency, our approach provides a library of corrective actions. We have implemented our approach as an extension to the IntensiVE tool suite which uses the SOUL logic meta programming language to define its intensional views. For each kind of relationship between intensional views supported by IntensiVE, we have applied our approach on an example taken from the implementation of the tool suite itself —thus illustrating the diagnosis and correction process supported by our approach.

Acknowledgements

We would like to acknowledge Johan Brichau for his invaluable comments and suggestions. Andy Kellens is funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). This work was partially funded by the Interuniversity Attraction Poles program of the Belgian Science Policy Office and the STADiUM project of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

References

[1] Albin-Amiot, H., 2001. JavaXL, a Java source code transformation engine. Tech. Rep. 2001-INFO, École des Mines de Nantes.

[2] Balzer, R., 1991. Tolerating inconsistency. In: Proceedings of the International conference on Software engineering (ICSE). IEEE Computer Society, pp. 158–165.

[3] Bauer, C., King, G., 2006. Java Persistence with Hibernate. Manning Publications Co., Greenwich, CT, USA.

[4] Beck, K., 1997. Smalltalk Best Practice Patterns. Prentice Hall.

[5] Besnard, P., Hunter, A., 1995. Quasi-classical logic: Non-trivializable classical reasoning from incosistent information. In: Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU). Springer-Verlag, pp. 44–51.

[6] Black, A. P., Ducasse, S., Nierstrasz, O., Pollet, D., 2009. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland.

[7] Brichau, J., De Roover, C., Mens, K., 2007. Open unification for program query languages. In: Proceedings of the International Conference of the Chilean Computer Science Society (SCCC). IEEE Computer Society, pp. 92–101.

[8] Brichau, J., Kellens, A., Castro, S., D'Hondt, T., April 2010. Enforcing structural regularities in software using IntensiVE. Science of Computer Programming: Experimental Software and Toolkits (EST 3) 75 (4), 232–246.

[9] Castro, S., Brichau, J., Mens, K., 2009. Diagnosis and semi-automatic correction of detected design inconsistencies in source code. In: International Workshop on Smalltalk Technologies (IWST). ACM, pp. 8–17.

[10] Castro, S., Mens, K., Brichau, J., 2008. Towards a taxonomy of tools for documenting code design. In: Working Session on Query Technologies and Applications for Program Comprehension (QTAPC), collocated with the International Conference on Program Comprehension (ICPC).

[11] CheckStyle, December 2006. Checkstyle. Http://checkstyle.sourceforge.net.

[12] Cole, B., Hakim, D., Hovemeyer, D., Lazarus, R., Pugh, W., Stephens, K., 2006. Improving your software using static analysis to find bugs. In: Companian to the Conference on Object-oriented programming systems, languages, and applications (OOPSLA). ACM, pp. 673–674.

[13] Console, L., Sapino, M. L., Dupré, D. T., 1995. The role of abduction in database view updating. Journal of Intelligent Information Systems 4 (3), 261–280.

[14] Coplien, J., 1992. Advance C++ Programming Styles and Idioms. Addison-Wesley.

[15] De Roover, C., August 2009. A logic meta programming foundation for example-driven pattern detection in object-oriented programs. Ph.D. thesis, Vrije Universiteit Brussel.

[16] De Roover, C., Michiels, I., Gybels, K., Gybels, K., D'Hondt, T., 2006. An approach to high-level behavioral program documentation allowing lightweight verification. In: Proceedings of the International Conference on Program Comprehension (ICPC). IEEE Computer Society, pp. 202–211.

[17] Flach, P., 1994. Simply logical: intelligent reasoning by example. John Wiley & Sons, Inc., New York, NY, USA.

[18] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[19] Guéhéneuc, Y.-G., 2007. Ptidej: A flexible reverse engineering tool suite. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE Computer Society, pp. 529–530.

[20] Hintikka, J., 1998. What is abduction? the fundamental problem of contemporary epistemology. In: Transactions of the Charles S. Peirce Society. Vol. 34. pp. 503–533.

[21] Hunter, A., Nuseibeh, B., 1998. Managing inconsistent specifications: reasoning, analysis, and action. ACM Transactions on Software Engineering and Methodology 7 (4), 335–367.

[22] Inoue, K., Sakama, C., 1995. Abductive framework for nonmonotonic theory change. In: Proceedings of the International Joint Conferences on Artificial intelligence (IJCAI). Morgan Kaufmann, pp. 204–210.

[23] Inoue, K., Sakama, C., 1998. Specifying transactions for extended abduction. In: Proceedings of the International Joint Conferences on Artificial intelligence (IJCAI). Morgan Kaufmann, pp. 394–405.

[24] Johnson, S., 2007. Lint. http://www.jutils.com/.

[25] Kakas, A. C., Mancarella, P., 1990. Database updates through abduction. In: Proceedings of the International conference on very large databases (VLDB). Morgan Kaufmann, pp. 650–661.

[26] Kakas, R. K. A., Tony, F., 1998. The role of abduction in logic programming. In: gabbay, C. H. D., Robinson, J. (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming. Oxford University Press.

[27] Kellens, A., 2007. Maintaining causality between design regularities and source code. Ph.D. thesis, Vrije Universiteit Brussel.

[28] Kellens, A., De Schutter, K., D'Hondt, T., Jorissen, L., Van Passel, B., 2009. Cognac: A framework for documenting and verifying the design of cobol systems. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR). IEEE Computer Society, pp. 199–208.

[29] Knodel, J., Muthig, D., Rost, D., 2008. Constructive architecture compliance checking - an experiment on support by live feedback. In: Proceedings of the International Conference on Software Maintenance (ICSM). IEEE Computer Society, pp. 287–296.

[30] Mens, K., Kellens, A., 2006. IntensiVE, a toolsuite for documenting and testing structural source-code regularities. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR). IEEE Computer Society, pp. 239–248.

[31] Mens, K., Michiels, I., Wuyts, R., 2001. Supporting software development through declaratively codified programming patterns. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE). Knowledge Systems Institute, pp. 236–243.

[32] Menzies, T., 1996. Applications of abduction: knowledge-level modelling. International Journal on Human Computer Studies 45 (3), 305–335.

[33] Minsky, N., 1996. Law-governed regularities in object systems; part 1: Principles. Theory and Practice of Object Systems (TOPAS) 2 (4), 283–301.

[34] Murphy, G. C., Notkin, D., Sullivan, K., 1995. Software reflexion models: Bridging the gap between source and high-level models. In: Proceedings on the symposium on Foundations of software engineering (FSE). ACM, pp. 18–28.

[35] Nuseibeh, B., Easterbrook, S., 2000. Requirements engineering: a roadmap. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, pp. 35–46.
[36] Nuseibeh, B., Russo, A., 1999. Using abduction to evolve inconsistent requirements specification. Australasian Journal of Information Systems (AJIS) 6 (2).
[37] Paavola, S., 2004. Abduction as a logic and methodology of discovery: the importance of strategies. Foundations of Science 9 (3), 267+.
[38] Pierce, C. S., 1935. The Collected Papers of Charles Sanders Peirce. Harvard University Press.
[39] Pierce, C. S., 1955. Abduction and induction. In: Buchler, J. (Ed.), Philosophical Writings of Pierce. Dover Books, New York, pp. 150–156.
[40] Poole, D., 1988. A logical framework for default reasoning. Artificial Intelligence 36 (1), 27–47.
[41] Reiter, R., April 1987. A theory of diagnosis from first principles. Artificial Intelligence 32 (1), 57–95.
[42] Russo, A., Miller, R., Nuseibeh, B., Kramer, J., 2000. An abductive approach for handling inconsistencies in SCR specifications. In: Proceedings of the Workshop on Intelligence Software Engineering (WISE), collocated with the International Conference on Software Engineering (ICSE).
[43] Russo, A., Nuseibeh, B., 2000. On the use of logical abduction in software engineering. In: Chang, S. K. (Ed.), Handbook on Software Engineering and Knowledge Engineering. World Scientific Publishing Corporation.
[44] Satoh, K., 1998. Adding and deleting pollution marker by abductive logic programming. In: Proceedings of the Asia-Pacific Workshop on Intelligent Software Engineering (APWISE), collocated with the Pacific Rim International Conference on Artificial Intelligence (PRICAI). pp. 48–53.
[45] Satoh, K., 1998. Computing minimal revised logic program by abduction. In: Proceedings of the International Workshop on the Principles of Software Evolution (IWPSE), collocated with the International Conference on Software Engineering (ICSE). pp. 177 – 182.
[46] Satoh, K., 2000. Consistency management in software engineering by abduction. In: Proceedings of the Workshop on Intelligent Software Engineering (WISE), collocated with the International Conference on Software Engineering (ICSE). pp. 90–99.
[47] Semmle Ltd., 2010. SemmleCode. http://semmle.com/.
[48] smacchia.com S.A.R.L, 2010. NDepend. http://www.ndepend.com/.
[49] Sullivan, P., 1991. On falsification interpretation of peirce. Transactions of the Charles S. Peirce Society 27, 197–219.
[50] Wuyts, R., 2001. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. Ph.D. thesis, Vrije Universiteit Brussel.
[51] Yu, C. H., April 1994. Abduction? deduction? induction? is there a logic of exploratory data analysis? Tech. rep., Annual Meeting of American Educational Research Association.
[52] Zave, P., Jackson, M., 1997. Four dark corners of requirements engineering. Transaction on Software Engineering Methodology 6 (1), 1–30.
[53] Zisman, A., Kozlenkov, A., 2001. Knowledge base approach to consistency management of UML specifications. In: Proceedings of the International Conference on Automated Software Engineering (ASE). IEEE Computer Society, p. 359.