Electronic Communications of the EASST
Volume 3 (2006)

Proceedings of the
Third Workshop on Software Evolution
through Transformations:
Embracing the Change
(SeTra 2006)

An Approach to Invariant-based Program Refactoring

Tiago Massoni, Rohit Gheyi and Paulo Borba

11 pages

Guest Editors: Jean-Marie Favre, Reiko Heckel, Tom Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

An Approach to Invariant-based Program Refactoring

Tiago Massoni[1], **Rohit Gheyi**[2] **and Paulo Borba**[3]

[1]tiago@dsc.upe.br,
Polytechnic School, University of Pernambuco
Recife, Brazil

[2]rg@cin.ufpe.br,
[3]phmb@cin.ufpe.br
Federal University of Pernambuco
Recife, Brazil

Abstract: Refactoring tools include checking of an object-oriented program for the fulfillment of preconditions, for ensuring correctness. However, program invariants – semantic information about classes and fields assumed valid during program execution – are not considered by this precondition checking. As a result, applicability of automated refactorings is constrained in these cases, as refactorings that would be applicable considering the invariants get rejected, usually requiring manual changes. In this paper, we describe initial work on the use of program invariants (declared as code annotations) to increase applicability of automated refactoring. We propose an approach that uses primitive program transformations that employ the invariant to make the program syntactically amenable to the desired refactoring, before applying the refactoring itself.

Keywords: program invariants, refactoring

1 Introduction

A popular technique for dealing with evolution-related problems is *refactoring* [Fow99, MT04], which improves software structure while preserving behavior to better support adaptations or additions. The practice of refactoring has been improved by supporting tools, avoiding manual work and increasing trust on behavior preservation. Usually a catalog of refactorings is offered, from which users can choose the desired transformation for the problem in context (for instance, push down fields of a class to a subclass, or introduction and renaming of classes and fields). These automated refactorings present *preconditions* that are checked against the code subject to refactoring, in order to ensure correctness.

Such precondition checking involves analysis of static information from the program – declarations and statements – enforcing strong conditions that limit the applicability of refactorings. While being effective to ensure safe refactorings, it leads to prevention of refactoring on programs that would be eligible if some semantic assumptions about the program behavior were taken into consideration. In these cases, extra manual refactoring is required, minimizing the benefits of using refactoring tools. Examples of such refactorings are presented in Section 2.

Semantic assumptions about the program can be expressed as *program invariants*, which consist of predicates about the state of objects and their inter-relationships, assumed valid throughout every possible program execution. In this paper, we describe how invariants can be used to increase the applicability of some automated refactorings (*invariant-based refactoring*). We indicate that if certain types of invariants are assumed, transformations based on these invariants can be applied to programs that would not be eligible for refactoring using the current tools. Programmers may provide invariants on classes and their fields as *code annotations*, or invariants may be discovered by existing analysis tools.

We propose an approach for using a sequence of *primitive program transformations* that employ invariants for making the program syntactically amenable to the desired refactoring, before applying the refactoring itself. As the program initially does not satisfy the preconditions, we use the invariants as a basis for applying some auxiliary program transformations that results in a program that can be subject to the desired automated refactoring. As a consequence, programs not fulfilling preconditions may be automatically refactored based on information about their behavior. By that, refactoring tools are applicable to more programs. Besides using code annotations, this approach can be applied in combination with program verification tools for discovering invariants from programs, using static or dynamic analysis.

2 Problem Statement

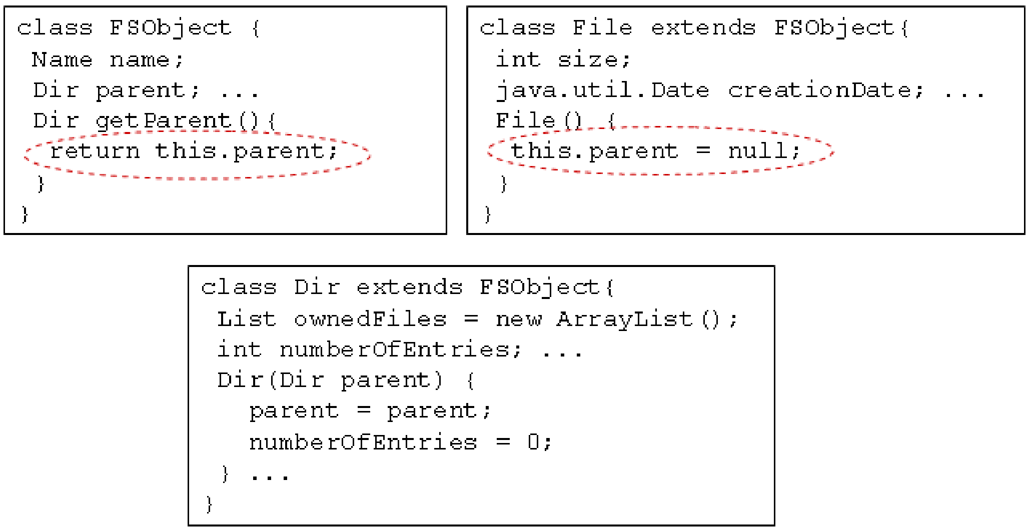
In this section, we make the case for invariant-based refactoring, by describing examples of refactoring tool limitations related to their precondition checking. Figure 1 shows a partial Java program representing a file system. The superclass `FSObject` defines a general representation for file system objects – files and directories – being specialized in correspondent subclasses. The `parent` field, declared into `FSObject`, defines the parent directory for each directory object.

Suppose that it is known that the `parent` field is always null for non-directory objects, such as files. In this context, it is desirable to apply a refactoring for *pushing down* `parent` to the `Dir` class [Fow99]. Refactoring tools, such as Eclipse, offer this refactoring in their catalog; selecting `parent`, the tool performs all related changes to apply this refactoring.

However, some of the previous accesses to the field do not allow the application of the refactoring. As indicated in the highlighted statements of Figure 1, accesses within `FSObject` and `File` would be invalid after moving the field to `Dir`. Nevertheless, `parent` does not make sense to non-directory objects – for example, `File` only allows `null` assignments to the inherited field –, and this semantic information is not taken into account by refactoring tools. This refactoring cannot be applied correctly in existing tools; otherwise, the resulting program would present typing errors (this situation will recur in other statically typed languages).

3 Program Invariants

Reasoning about design of object-oriented programs usually relies on a number of *object invariants*, which represent consistency conditions on the program's objects and its data fields that must be maintained throughout the execution of the program [LM04]. This information may be stated



```
class FSObject {
 Name name;
 Dir parent; ...
 Dir getParent(){
  return this.parent;
 }
}
```

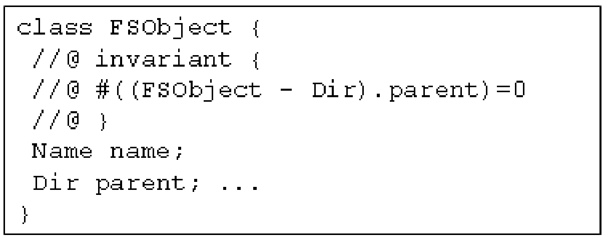
```
class File extends FSObject{
 int size;
 java.util.Date creationDate; ...
 File() {
  this.parent = null;
 }
}
```

```
class Dir extends FSObject{
 List ownedFiles = new ArrayList();
 int numberOfEntries; ...
 Dir(Dir parent) {
   parent = parent;
   numberOfEntries = 0;
 } ...
}
```

Figure 1: File system implementation.

in separate artifacts, such as object models [LG01], or integrated into programming languages, using *annotation languages*. Code annotations can handle the complexities of object-oriented code, also being directly compilable into runtime assertions.

Examples of annotation languages for Java include the Java Modeling Language (JML) [BCC+05] and Alloy Annotation Language (AAL) [KMJ02]. We illustrate the use of annotations by showing object invariants for `FSObject` and `Dir`, in Figure 2. The invariants are based on simple first-order logic based on AAL. In `FSObject`, the invariant states that file system objects, except directories, have no parent directories; `FSObject-Dir` yields all `FSObject` instances that are not `Dir` instances. The set of dereferences of the `parent` attribute from these instances is always empty (# denotes set cardinality).



```
class FSObject {
 //@ invariant {
 //@ #((FSObject - Dir).parent)=0
 //@ }
 Name name;
 Dir parent; ...
}
```

Figure 2: Object invariants as code annotations.

Code annotations are provided in at least three different ways. Users may add annotations as supplementary design information that may help program documentation and analysis. For instance, advanced static analysis can be applied to programs annotated with invariants [FLL+02].

Similarly, invariants may be transferred to the program by abstract models *in conformance*. For instance, the invariant from Figure 2 could have been defined in a structural model (UML class diagrams with constraints) which the program conforms to. Conformance implies that if the model constrains the `parent` relationship to directories only, the classes implementing the involved concepts must follow this constraint.

Furthermore, likely invariants may also be discovered from several executions of a program, as seen with Daikon tool support [ECGN01]. It consists in a program analysis that generalizes over observed values to assume program properties, used in testing, verification and bug detection. In this case, user intervention for dealing with invariants is minimized.

4 Approach for Invariant-based Refactoring

In Section 2, we exemplified a program that would be eligible to automated refactorings if semantic information was considered. In this section, we show how program invariants that represent this semantic information can be used as a basis for increasing the applicability of such automated refactorings. We first show a systematic method for applying behavior-preserving transformations to programs (Section 4.1), that will aid making programs subject to refactoring. These transformations build a foundation for our approach, as described in Section 4.2. Other examples of invariants that could be applied accordingly are defined in Section 4.3.

4.1 Primitive Program Transformations

Refactoring must preserve the observable behavior of a program. A way to facilitate mechanization of refactorings in tool support is to adopt an *algebraic* method, in which a refactoring is made of a sequence of behavior-preserving transformations [Opd92]. One classical approach for defining these transformations can be *primitive* laws [HHJ+87] relating language constructs. Easily mechanized by term rewriting, these laws are immediately available as a framework for transforming programs. In addition, primitive laws may be composed for deriving large-grained transformations that preserve semantics of programs. For object-oriented programming, an extensive set of laws has been defined for the Refinement Object-Oriented language (ROOL) [B+04], which corresponds to a subset of Java.

Although defined for a simplified language, most laws can be leveraged to Java. The most critical restriction is on its *copy semantics*, rather than a reference semantics [B+04]. In ROOL, objects are treated as primitive values (records), consequently presenting no pointers. The laws shown in this section do not deal with object sharing; however, for laws that depend on sharing to be correct, an additional property of *confinement* must be ensured. In fact, the laws must be revised in order to deal with reference semantics. In this section, the laws are presented following the Java syntax, for simplicity.

As an example of primitive law, the following moves a field up (applying from left to right) or down (right to left) to super or subclass, respectively, which is put in practice by refactoring tools. Each law denotes two transformations, as it defines equivalence. The provisos (preconditions) ensure that the transformations denoted by the law preserve semantics. The equivalence is valid within a context of class declarations *cds* and a main method *c*. The symbol '$(\rightarrow)$' before the first proviso indicates it is only required for applications of this law from left to right. On the other hand, '$(\leftarrow)$' is used when a proviso is necessary only for applying a law from right to left.

Law 1 $\langle$*move field to superclass*$\rangle$

$$\boxed{\begin{array}{l} \textbf{class}\ B\ \textbf{extends}\ A\ \{ \\ \quad ads \\ \quad ops \\ \} \\ \textbf{class}\ C\ \textbf{extends}\ B\ \{ \\ \quad a : T; ads' \\ \quad ops' \\ \} \end{array}} =_{cds,c} \boxed{\begin{array}{l} \textbf{class}\ B\ \textbf{extends}\ A\ \{ \\ \quad a : T; ads \\ \quad ops \\ \} \\ \textbf{class}\ C\ \textbf{extends}\ B\ \{ \\ \quad ads' \\ \quad ops' \\ \} \end{array}}$$

provided
$(\rightarrow)$ The field name a is not declared by the subclasses of B in cds;
$(\leftarrow)$ $D.a$, for any $D \leq B$ and $D \nleq C$, does not appear in cds, c, ops, or ops'.

The *ads* identifier represents field declarations in the class, while *ops* stands for the declaration of methods and constructors. The notation $B.a$ denotes uses of a through expressions whose static type is exactly B (for instance, an expression yielding an object from class B, strictly). To denote that B is a subclass of A, we write $B \leq A$. The second proviso above precludes an expression such as `this.a` from appearing in *ops*, but does not preclude `this.c.a`, for a field $c : C$ declared in B. The last expression is valid in *ops* no matter whether a is declared in B or in C.

Several laws for commands and expressions complement laws for object-oriented constructs. For instance, the next law allows us to introduce type casts to expressions, as long as the type of the expression is consistent with the cast, given the type context (denoted by $\triangleright$).

Law 2 $\langle$*introduce trivial cast in expressions*$\rangle$
If $cds, A \triangleright e : C$, then $cds, A \triangleright e = (C)e$.

For ensuring that the program's state before a given statement fulfills a given invariant, we make use of Java assertions. Each assertion contains a boolean expression that you believe will be true when the assertion executes. If it is not true, the system will throw an error. We can, for example, extract assertions from guards, as stated by the following law, where ψ_i denotes a boolean formula.

Law 3 $\langle$*assertion condition*$\rangle$
$\textbf{if}\ (\psi_i)\ \{\ c_i\ \} = \textbf{if}\ (\psi_i)\ \{\ assert\ (\psi_i);\ c_i\ \}$

4.2 An Approach for Invariant-based Refactoring

We now describe an approach for applying behavior-preserving transformations for refactoring programs based on invariants. A type of invariant is identified; for that invariant, the tool can apply a predefined sequence of primitive laws of programming which we call *strategy*. Strategies possess two key properties: (1) they must rewrite programs for updating statements, using invariants, before the desired refactoring; (2) they preserve program behavior, which entails from the application of laws.

Checking the program from Figure 1 in current refactoring tools involves type declarations for ensuring correctness. As the `parent` field is to be pushed down to a particular subclass, accessing `parent` from a reference whose type is not of that subclass would be invalid. On the other hand, an intuitive analysis of the program in the light of the invariant from Figure 2 shows some interesting aspects about those fragments. The invariant guarantees that the `parent` field will be `null` for any object references whose type is `FSObject` or its subclasses, except `Dir`.

Strategies are illustrated by refactorings applied to the program from Section 2. For pushing down the `parent` field, Law ⟨*move field to superclass*⟩ may be a straightforward option to refactor the program. However, the highlighted statements in Figure 1 clearly prohibit the application of the law, as they make the required provisos invalid. Our aim is to apply other primitive laws that acquire value from this information. The following derivation is related to the method `getParent`, within which `parent` is read. This derivation is illustrative for the example; in fact, this strategy can be generalized for programs in a similar context.

Step 1. Within `FSObject`, the value of the `this` identifier obeys the following condition: `this instanceof Dir || !(this instanceof Dir)`. We express this condition by the following `if` statement. Using Law ⟨*assertion condition*⟩, we can change the body of `getParent` by extracting assertions from each branch.

```
Dir getParent(){
 if (this instanceof Dir){
  assert (this instanceof Dir);
  return this.parent;
 } else{
  assert !(this instanceof Dir);
  return this.parent;
 }
}
```

Step 2. Concerning the first branch, we can introduce a cast to the assignment, using Law ⟨*introduce trivial cast in expressions*⟩.

```
Dir getParent(){
 if (this instanceof Dir){
  return ((Dir)this).parent;
 } else{
  assert !(this instanceof Dir);
  return this.parent;
 }
}
```

Step 3. In the second branch, we now can use the invariant from `FSObject`, defined as `#((FSObject-Dir).parent)=0`. It is introduced as an assertion conjoined with the previous one, as follows (translated to Java as `this instanceof Dir || this.parent==null`).

```
Dir getParent(){
 if (this instanceof Dir){
  return ((Dir)this).parent;
 } else{
  assert (this instanceof Dir || this.parent==null &&
                               !(this instanceof Dir));
  return this.parent;
 }
}
```

Step 4. Still in the second branch, we use simple logic rules for simplifying the assertion (`!A && A == false`).

```
Dir getParent(){
 if (this instanceof Dir){
  return ((Dir)this).parent;
 } else{
  assert (this.parent==null);
  return this.parent;
 }
}
```

Step 5. As stated in the assertion, the program state defines the value read by the assignment (already `null` before the command), resulting in the following body for *getParent*.

```
Dir getParent(){
 if (this instanceof Dir){
  return ((Dir)this).parent;
 } else{
  return null;
 }
}
```

A similar derivation can be developed in the constructor of `File`, assigning `null` to `parent`:

Step 1. Since the command is within `File`, we can introduce the following assertion.

```
File(){
 assert (this instanceof File);
 this.parent=null;
}
```

Step 2. The program invariant (`this instanceof Dir || this.parent==null`), is then introduced to the assertion.

```
File(){
 assert (this instanceof Dir || this.parent==null &&
                              this instanceof File);
 this.parent=null;
}
```

Step 3. The assertion can now be simplified accordingly.

```
File(){
 assert (this.parent==null && this instanceof File);
 this.parent=null;
}
```

Step 4. The command has no effect over the state (`this.parent` possesses a constant value before and after) and no other variable is changed. Hence, the assignment (along with the assertion) can be removed with no impact on the program's behavior.

After the application of this strategy, the program now can be subject to Law ⟨*move field to superclass*⟩, from right to left. The same could have been done to other similar occurrences of `parent` in the program. Consequently, these can be generalized for any other program presenting this type of invariant. The general sequence of transformations before the actual move operation constitutes a strategy with the aid of the program invariant, to be applied in conjunction with the refactoring tool.

4.3 Other invariants

Similar strategies can be defined for several types of invariants. Some of the invariants we investigated are summarized next:

- **Remove field.** In general automated refactorings only remove fields when they are not used anywhere in the program. A strategy can prepare programs that do not present this property – although removal of the given field is desirable – by replacing all reads from the field to be removed by the correspondent value given by an invariant. For instance, if the invariant `this.newField=this.oldField` is provided for the class declaring the field, and `oldField` is to be removed, we can use the invariant to replace reads from `oldField` by the corresponding expression (`newField`), eliminating writings. This is possible since no other variable depends on this field (all reads have been removed).

- **Replace array field by single variable field.** A field can be declared as an array even though a design assumption defines the field as empty or holding only one element. This is due to planned additions that did not come about, as for example accounts in a bank that were defined with the policy of holding at most one credit card, and this assumption did not change in the future. In this case, we can change its declaration and statements to a single variable, given the invariant on the multiplicity of the field. For instance, an array field `var`, in the presence of an invariant `#this.var=1` on its cardinality, can store this single value on a variable; laws can be applied to change the statement to use the variable (for instance, `this.var[0] = a` becomes `this.var = a`. The reverse transformation (variable to array) can be applied as well.

5 Conclusions and Future Work

In this paper, we describe an approach for automatic refactorings that assumes program invariants for offering more applicability to refactoring tools, avoiding some manual adjustments. Invariants – declared as code annotations – provide semantic information about classes and their fields, which is used to refactor the program in primitive steps – laws of programming – offering a greater degree of applicability.

There are several other open questions. For instance, it is not clear how invariants will be automatically identified by a refactoring tool for the application of specific refactorings. Our intuition is that catalogs of program refactorings could be extended with improvements based on invariants, conditionally applied based on a set of found invariants. Also, other types of invariants must be investigated, in order to establish a more general notion of invariants that can aid automated refactorings. These accomplishments are critical to incorporating invariant-based refactorings in tool support.

We plan also to extend this approach to consider other types of annotations, such as pre- or post-conditions of methods. We believe that these invariants may help more powerful refactorings involving methods, such as the Extract Method refactoring [Fow99]. Other promising research topic is exploring invariants not only for refactorings, but also general evolution transformations (adding a new feature to the program, for example).

In a previous work [MGB05], we propose an approach for refactoring object models (such as UML class diagrams with OCL invariants [B+99, W+03]) and programs, as shown in Figure 3, which is an application of invariant-based refactoring. A refactoring is applied to the object model, and a program in conformance with the model is automatically refactored accordingly. The program refactorings are applied automatically from the semantic information given by models.

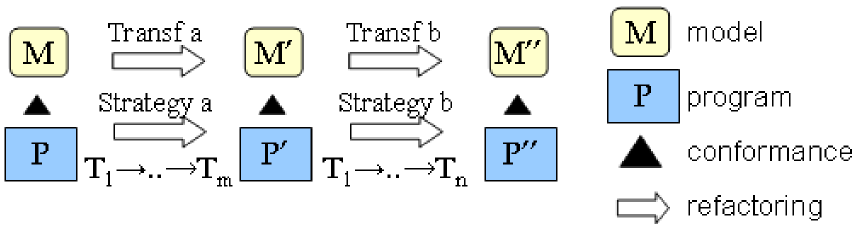


Figure 3: Model-driven Refactoring.

We consider object model refactoring as a composition of primitive semantics-preserving transformations. Each model transformation applied to the model triggers the application of a strategy to the source code. The main idea behind strategies is the assumption that the original program is in conformance with the model, implying that all model invariants are guaranteed to be true in the program. Therefore, the same principle of invariant-based refactoring is applied, in which predefined model transformations provide the original model to which the transformation is applied, so the invariants that are considered true in the program are known in advance. The program transformation is applied independently, although based on the model transformation;

this scenario avoids the problems related to round-trip engineering tools, in which programs are generated from models, and vice-versa. This is certainly a useful formal investigation for modern development methodologies, such as Model-driven Architecture [K+03].

Acknowledgements: We'd like to thank the anonymous referees for useful suggestions, besides members of the Software Productivity Group.

Bibliography

[B+99] G. Booch et al. *The Unified Modeling Language User Guide*. Object Technology. Addison Wesley, 1999.

[B+04] P. Borba et al. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming* 52:53–100, October 2004.

[BCC+05] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer*, 2005.

[ECGN01] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering* 27(2):1–25, 2001.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended Static Checking for Java. *ACM SIGPLAN Notices* 37(5):234–245, 2002.

[Fow99] M. Fowler. *Refactoring—Improving the Design of Existing Code*. Addison Wesley, 1999.

[HHJ+87] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, B. A. Sufrin. Laws of Programming. *Communications of the ACM* 30(8):672–686, 1987.

[K+03] A. Kleppe et al. *MDA Explained: the Practice and Promise of The Model Driven Architecture*. Addison Wesley, 2003.

[KMJ02] S. Khurshid, D. Marinov, D. Jackson. An Analyzable Annotation Language. In *Proceedings of the 17th OOPSLA*. Pp. 231–245. ACM Press, 2002.

[LG01] B. Liskov, J. Guttag. *Program Development in Java*. Addison Wesley, 2001.

[LM04] K. R. M. Leino, P. Müller. Object Invariants in Dynamic Contexts. In *ECOOP*. Pp. 491–516. 2004.

[MGB05] T. Massoni, R. Gheyi, P. Borba. A Model-driven Approach to Formal Refactoring. In *Companion to the OOPSLA 2005*. Pp. 124–125. USA, October 2005.

[MT04] T. Mens, T. Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30(2):126–139, February 2004.

[Opd92] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[W$^+$03] J. Warmer et al. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, second edition, 2003.