Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 184 (2007) 39–61
www.elsevier.com/locate/entcs

Using CafeOBJ to Mechanise Refactoring Proofs and Application

Antonio Carvalho Júnior[1]

Departamento de Ciência da Computação e Estatística
Universidade Federal de Sergipe (UFS)
CEP 41900-000 – São Cristóvão – SE, Brazil

Leila Silva[2]

Departamento de Ciência da Computação e Estatística
Universidade Federal de Sergipe (UFS)
CEP 41900-000 – São Cristóvão – SE, Brazil

Márcio Cornélio[3]

Departamento de Sistemas Computacionais
Escola Politécnica – Universidade de Pernambuco (UPE)
Rua Benfica, 455 – 50750-410, Recife – PE, Brazil

Abstract

In this paper we show how rewriting systems, in particular CafeOBJ, can be used to automatically prove refactoring rules. In addition, a small case study that illustrates the application of a refactoring rule in an arbitrary program is also developed. Our approach is based on a sequential object-oriented language of refinement (ROOL) similar to Java. We have implemented the ROOL grammar in CafeOBJ, as well as the laws that define its semantics. Each refactoring rule is derived by the application of these laws, in a constructive way. The refactorings are also implemented in CafeOBJ, allowing the reduction of an arbitrary program.

Keywords: Rewriting Systems, Refactorings, CafeOBJ

1 Introduction

Changes are common to software. Many practitioners recognize that changing an object-oriented software is easier than conventional ones [15]. Refactoring is the

[1] Email: acjr@dcce.ufs.br
[2] Email: leila@ufs.br
[3] Email: mlc@dsc.upe.br

doi:10.1016/j.entcs.2007.03.014

activity of modifying a software system by preserving the external behaviour, perceived by the user. In fact, just the internal software structure is affected, by for example, moving attributes and methods between classes.

In [10,21] several refactorings are introduced in a rather informal way. Opdyke [19] formalises refactorings to the degree that they can be encoded in tools. Cinnéide and Nixon [6] present a semi-formal approach to demonstrate behaviour preservation for design patterns transformations, defined in terms of refactorings. Lano *et al* [12] formally justify design patterns by relating two set of classes, the "before" and "after" systems. The "after" system consists of a collection of classes organised according to patterns. Selected axioms are used to prove that the "after" system is an extension of the "before" one. Cornélio [8] adopts a transformational approach, that is constructively based on rules, to formalise several refactorings introduced previously by Fowler [10].

The work proposed by Cornélio [8] is based on transformation rules between meta-programs in ROOL (Refinement Object-Oriented Language) [3], which is a subset of sequential Java [1] with classes, inheritance, visibility control for attributes, dynamic binding and recursion. Refactorings are described in ROOL as rules, relating the meta-program on the left-hand side to another one on the right-hand side. Each rule is derived from programming and refinement laws, also expressed in ROOL.

In this paper we show how the refactoring rules can be proved using the rewriting system CafeOBJ [18]. Moreover, a case study showing the application of the refactorings is also developed. In this way, this work complements the formal rigour of [8], as manual proofs can easily hold mistakes. In addition, the results achieved suggest that rewriting systems can be used as supporting tools for the construction of refactoring environments.

Although CafeOBJ does not seem to be used in the context of refactorings, the use of rewriting systems to implement algebraic reduction strategies have already been proposed. Lira *et al* [14] use Maude [7] to implement a reduction strategy for object-oriented languages and Silva *et al* [20] implement in CafeOBJ a reduction strategy in the context of hardware/software partitioning.

This paper is organised as follows. Section 2 introduces ROOL, the language used to formalise the refactorings. In Section 3 we present three refactoring rules used to illustrate the work here developed. The rewriting system CafeOBJ is briefly introduced in Section 4. Section 5 presents the mechanisation of the refactoring proofs, as well as the implementation of two refactorings in CafeOBJ. A case study developed to validate the work is described in Section 6. Finally, Section 7 gives the conclusions and directions for future work.

2 ROOL and Laws

ROOL [4,3] is an object-oriented language based on sequential Java. It allows reasoning about object-oriented programs and specifications, as both kind of constructs are mixed in the style of Morgan's refinement calculus [16,17]. The semantics of ROOL, as usual for refinement calculi, is based on weakest preconditions. The imper-

ative constructs of ROOL are based on the language of Morgan's refinement calculus [16], which is an extension of Dijkstra's language of guarded commands [9]. The meaning of ROOL constructs is expressed by algebraic laws, in the style of [16].

A program $cds \bullet c$ in ROOL is a sequence of classes cds followed by a main command c. Classes are declared as in the following example, where we define a class *Employee*.

class *Employee* **extends object**
  pri *salary* : **int**;
  meth *getSalary* $\widehat{=}$ (**res** r : **int** $\bullet$ r := **self**.*salary*)
  meth *setSalary* $\widehat{=}$ (**val** s : **int** $\bullet$ **self**.*salary* := s)
  new $\widehat{=}$ **self**.*salary* := 0
end

Classes are related by single inheritance, which is indicated by the clause **extends**. The class **object** is the default superclass of classes. So, the **extends** clause could have been omitted in the declaration of *Employee*. The class *Employee* includes a private attribute named *salary*; this is indicated by the use of the **pri** qualifier. Attributes can also be protected (**prot**) or public (**pub**). ROOL allows only redefinition of methods which are public and can be recursive; they are defined using procedure abstraction in the form of Back's parameterized commands [2,5]. A parameterised command can have the form **val** $x : T \bullet c$ or **res** $x : T \bullet c$, which correspond to the call-by-value and call-by-result parameter passing mechanisms, respectively. For instance, the method *getSalary* has a result parameter r, whereas *setSalary* has a value parameter s. Initialisers are declared by the **new** clause. ROOL also includes specification constructs from Morgan's refinement calculus [16], like the specification statement.

The laws of ROOL, mainly those related to object-oriented features, address context issues. We use $cds_1 =_{cds,c} cds_2$, where cds is a context of class declarations cds_1 and cds_2, and c is the main command to denote the equivalence of set of class declarations cds_1 and cds_2. This notation is just an abbreviation for the program equivalence $cds_1 cds \bullet c = cds_2 cds \bullet c$, which is fomalised in [4,3]. As an example, in what follows we present some laws of ROOL. We write '$(\rightarrow)$' when some conditions must be satisfied for the application of the law from left to right. We use '$(\leftarrow)$' to indicate the conditions that are necessary for applying a law from right to left. By writing '$(\leftrightarrow)$' we indicate the conditions necessary in both directions. Conditions are described in the **provided** clause of laws.

A complete set of ROOL laws can be found in [8]. Here we only introduce some laws of command and laws of classes, necessary to understand the mechanisation of the proofs, presented in Section 5. The explanations of these laws are extracted from [8].

Law 1 $\langle := - \lhd \rhd$ *right dist*$\rangle$ distributes an assignment over a list of conditional commands, provided e is total.

Law 1 $\langle := - \lhd \rhd$ *right dist*$\rangle$ If e is total, then
$le := e$; **if** $[]i \bullet \psi_i \rightarrow c_i$ **fi** = **if** $[]i \bullet \psi_i[e/le] \rightarrow (le := e; c_i)$ **fi** $\Box$

Law 2 $\langle var\ elim \rangle$ eliminates a variable that does not appear in c and Law 3 $\langle var\text{-} := final\ value \rangle$ removes an assignment at the very end of its scope, as this assignment introduces no effect.

Law 2 $\langle var\ elim \rangle$ If x is not free in c, then $\mathbf{var}\ x : T \bullet c\ \mathbf{end} = c$ □

Law 3 $\langle var\text{-} := final\ value \rangle$ If x is not free in c, then

$\mathbf{var}\ x : T \bullet c;\ x := e\ \mathbf{end} = \mathbf{var}\ x : T \bullet c\ \mathbf{end}$ □

Law 4 $\langle var\text{-} := initial\ value \rangle$ assigns an expression to a variable before its first use and Law 5 $\langle order\ independent\ assignment \rangle$ reorders assignments, if they are not data dependent. The symbol $\sqsubseteq$ means the standard refinement symbol.

Law 4 $\langle var\text{-} := initial\ value \rangle$ If e has no type errors, then

$\mathbf{var}\ x : T \bullet c\ \mathbf{end} \sqsubseteq \mathbf{var}\ x : T \bullet x := e;\ c\ \mathbf{end}$ □

Law 5 $\langle order\ independent\ assignment \rangle$ If x and y are not free in e_2 and e_1, respectively, then $(x := e_1; y := e_2) = (y := e_2; x := e_1)$ □

Law 6 $\langle method\ elimination \rangle$, allows the remotion of a method from a class if it is not called by any classes in cds, in the main command c, nor inside class C. To apply this law from right to left, the method m cannot be already declared in C nor in any of its superclasses or subclasses, so that we can introduce a new method in a class. The notation $b.m$ refers to calls to a method m via expressions whose static type is exactly b. The subclass relation is denoted by $\leq$. We write $B \leq A$ to denote that a class B is a subclass of a class A.

Law 6 $\langle method\ elimination \rangle$

class C **extends** D ads **meth** $m \mathrel{\widehat{=}} pc$ mts **end**	$=_{cds,c}$	**class** C **extends** D ads mts **end**

provided

($\rightarrow$) $B.m$ does not appear in cds, c nor in mts, for any B such that $B \leq C$.

($\leftarrow$) m is not declared in ops nor in any superclass or subclass of C in cds.

□

Law 7 $\langle move\ original\ method\ to\ superclass \rangle$, allows us to move an original method to a superclass. Between other requirements, this law demands that occurrences of **self** in methods to be moved are cast.

Law 7 $\langle move\ original\ method\ to\ superclass \rangle$

class B **extends** A ads mts **end** **class** C **extends** B ads' **meth** $m \mathrel{\widehat{=}} pc$ mts' **end**	$=_{cds,c}$	**class** B **extends** A ads **meth** $m \mathrel{\widehat{=}} pc$ mts **end** **class** C **extends** B ads' mts' **end**

provided
($\leftrightarrow$) (1) **super** and private attributes do not appear in pc; (2) m is not declared in any superclass of B in cds;
($\rightarrow$) (1) m is not declared in mts, and can only be declared in a class D, for any $D \leq B$ and $D \not\leq C$, if it has the same parameters as pc; (2) pc does not contain uncast occurrences of **self** nor expressions in the form $((C)\mathbf{self}).a$ for any private attribute a in ads';
($\leftarrow$) (1) m is not declared in mts'; (2) $D.m$, for any $D \leq B$, does not appear in cds, c, mts or mts'.

□

3 Refactoring Rules

A comprehensive set of refactoring rules which captures and formalises most of the refactorings informally introduced in [10] is presented in [8]. Refactoring rules are described by means of two boxes written side by side, along with **where** and **provided** clauses. We use the **where** clause, when necessary, to write abbreviations. The provisos for applying a refactoring rule are listed in the **provided** clause of the rules. Here we present three examples of refactoring rules. The first one allows extracting and inlining a method, and is used to illustrate the mechanisation of a refactoring proof in CafeOBJ. The second one introduces set and get methods and is used to show an implementation of a refactoring rule in CafeOBJ, as well as its application in an arbitrary example. The last onde moves a method from a class to another, and is used in the case study developed here.

3.1 Extract and Inline Method

Rule 1, when considered from left to right, coincides with the refactoring Extract Method presented by Fowler [10, p. 110], whereas the application in the reverse direction corresponds to the refactoring Inline Method [10, p. 117]. When applied from left to right, it turns a command c_2, which is present in a method m_1, into a new method m_2. Occurrences of the command c_2 in the original method m_1 are replaced by calls to the newly introduced method.

Rule 1 $\langle Extract/InlineMethod \rangle$

class A **extends** C ads; **meth** $m_1 \mathrel{\widehat{=}}$ $(pds_1 \bullet c_1[c_2[a]])$ mts **end**	$=_{cds,c}$	**class** A **extends** C ads; **meth** $m_1 \mathrel{\widehat{=}} (pds_1 \bullet c_1')$ **meth** $m_2 \mathrel{\widehat{=}} (pds_2 \bullet c_2[\alpha(pds_2)])$ mts' **end**

where

$c_1' \mathrel{\widehat{=}} c_1[\mathbf{self}.m_2(a)/c_2[a]]$

$mts \mathrel{\widehat{=}} mts'[c_2[a]/\mathbf{self}.m_2(a)]$

a a is the finite set of free variables of command c_2, not including attributes of class A;

provided

($\leftrightarrow$) (1) Variables in a have basic types; (2) Parameters in pds_2 must have the same types as those of variables in a;

($\rightarrow$) m_2 is not declared in mts nor in any superclass or subclass of A in cds;

($\leftarrow$) m_2 (1) is not recursively called; (2) $B.m_2$ does not appear in cds, c nor in mts, for any B such that $B \leq A$.

The meta-variable a represents a list containing the free variables that appear in the command c_2 of method m_1 which are not attributes of class A. On the left-hand side of this rule, $c_1[c_2[a]]$ represents the command c_1, which may have occurrences of the command c_2, which in turn may have occurrences of a. The class C that appears in the **extends** clause, and in the others that follow, is present in the sequence of class declarations cds or is the class **object**.

On the right-hand side of this rule, the method call $\mathbf{self}.m_2(a)$ replaces the occurrences of the command $c_2[a]$ in the command c_1 of method m_1; the resulting command is c_1'. In the command c_2 of method m_2, the variables indicated by a are replaced with the parameters $\alpha(pds)$, where $\alpha(pds)$ denotes the list of parameter names declared in pds. If a variable is only read in c_2, it could be passed as a value argument. A variable that is only written could be passed as a result argument. Variables that are both read and written must be passed as both value and result arguments. The free variables, represented by a, that are passed as arguments in the call to m_2 may involve all arguments of method m_1 as well as local variables that appear in c_2. Of course, all free variables that appear in c_2 must become parameters of m_2.

To apply this rule from left to right, the method m_2 must be new: not declared in a superclass of A, in A itself, nor in any of its subclasses. We also require the types of variables in a to be basic, and parameters in pds_2 must have the same types as those of a. Applying this rule from right to left replaces method calls to

m_2 with the body of this method and removes m_2 from class A. To apply this rule in this direction, there must be no recursive calls in the method m_2 (note that this is not a limitation in practice because c_2 can be a recursive command of the form **rec** $X \bullet c_3$ **end**). Also, the method m_2 cannot be called in cds, c nor in mts.

Before presenting the derivation of this refactoring, we need to introduce two results, proved in [8].

Lemma 1 (*method call elimination-**self***) replaces the call for a parameterised command which is the body of a local method with a call to the method itself. The symbol $\rhd$ in $cds, C \rhd c$ asserts that c is a command that can appear in the body of a method in class C, in the context of the sequence of class declarations cds.

Lemma 1 (*method call elimination-**self***) *Consider that the following class declaration*

class C **extends** D
ads
meth $m \mathrel{\widehat{=}} pc$
mts
end

is included in cds. Then $cds, C \rhd pc(e) = \mathbf{self}.m(e)$

□

Lemma 2 (*pcom value − argument*) transforms a single command into a parameterised one with a value argument.

Lemma 2 (*pcom value − argument*) $c = (\mathbf{val}\ vl : T \bullet c[vl/x])(x)$, *provided vl is not free in c, x is not on the left-hand side of assignments, it is not a result argument, x does not occur in the frame of specification statements in c, x is not a method call target, and* $x \neq error$

□

Derivation. We begin the derivation of Rule 1 with the class A that appears on the left-hand side of Rule 1. We assume that the required conditions for the application of this rule hold. However, it is not possible to have a fully general derivation, in the sense that we do not define a parameter for the method being extracted, rather we have to introduce a parameterised command along with a specific parameter passing mechanism in order to be able to conduct the proof. Moreover, in the derivation we consider that the set of variables a has just one element, a variable named a. Also, we consider that such variable is only read, implying that the value parameter passing mechanism is the one applicable for the parameter to be defined in the extracted method. The derivation for a result parameter is similar.

By using Law 6 ⟨*method elimination*⟩, from right to left, we introduce the following method m_2 in class A.

meth $m_2 \mathrel{\widehat{=}} (\mathbf{val}\ arg : T \bullet c_2[arg])$

This requires that the method to be introduced is not declared in the superclass of the class to which the law is applied, in the class itself nor in any of its subclasses. We can apply Law 6 ⟨*method elimination*⟩ because the refactoring rule requires the same conditions to be satisfied.

The command c_2 that is present in the method m_1 also appears in the method m_2. Our aim is to introduce a call to the method m_2. We introduce a parameterised command that is applied to the argument a by using Lemma 2 ($pcom\ value - argument$). The occurrences of variable a, in command c_2, are replaced with the parameter arg. This lemma requires that the argument that is applied to the parameterised command is not a method call target. Consequently, there is no sense in having as argument an object on which we cannot call a method inside the parameterised command. For this reason a is restricted to have basic type. By the application of Lemma 2 ($pcom\ value - argument$), we obtain the following parameterised command.

$(\mathbf{val}\ arg : T \bullet c_2[arg])(a)$

The parameterised command that occurs in command m_1 is the same as that of method m_2. By using Lemma 1 (*method call elimination-**self***), from left to right, we introduce in m_1 a call to m_2, obtaining the following class.

= Lemma 1 (*method call elimination-**self***)

class A
  ads;
  meth $m_1 \mathrel{\widehat{=}} (pds_1 \bullet c_1[\mathbf{self}.m_2(a)])$
  meth $m_2 \mathrel{\widehat{=}} (\mathbf{val}\ arg : T \bullet c_2[arg])$
  mts
end

This completes the derivation for a value parameter. The derivation for an arbitrary number of parameters is similar, but it involves the application of Law $\langle pcom\ merge \rangle$ [8], which merges two parameterised commands into one.

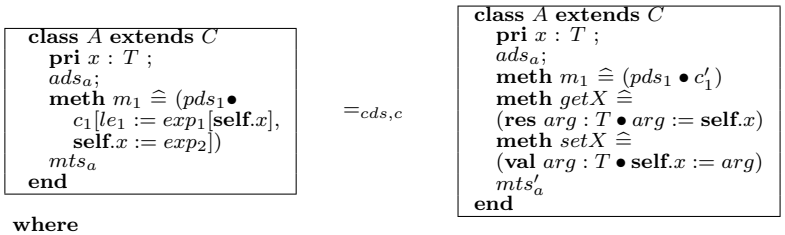
3.2 Self Encapsulate Field

Rule 2 $\langle SelfEncapsulateField \rangle$ introduces get and set methods for an attribute x declared in the class A. The derivation of this rule can be found in [8].

In method m_1 on the left-hand side of the rule, the attribute x appears in the expression exp_1 and there is also an assignment to this attribute. On the right-hand side of the rule, the occurrence of $\mathbf{self}.x$ in expression exp_1 is replaced by the local variable aux declared in m_1. This variable receives the result of the call to method $getX$. The assignment is accomplished by a call to method $setX$, passing by value the expression exp_2. These changes also occur in mts_a . To apply this rule from left to right, the methods $getX$ and $setX$ cannot be declared in the superclass of A, in A itself, nor in any of its subclasses. To apply this rule in the reverse direction, the methods $getX$ and $setX$ cannot be called in cds, c, or A.

In this rule, just one attribute is considered. To encapsulate fields inside a class implies in the application of this rule the same number of times as the number of attributes to be encapsulated.

Rule 2 $\langle SelfEncapsulateField \rangle$



where
$c'_1 \mathrel{\widehat{=}} c_1[\textbf{var}\ aux : T \bullet \textbf{self}.getX(aux);\ le_1 := exp_1[aux]\ \textbf{end},\ \textbf{self}.setX(exp_2)\ /\ le_1 := exp_1[\textbf{self}.x],\ \textbf{self}.x := exp_2]$
$mts'_a \mathrel{\widehat{=}} mts_a[\textbf{var}\ aux : T \bullet \textbf{self}.getX(aux);\ le_1 := exp_1[aux]\ \textbf{end}, \textbf{self}.setX(exp_2)\ /\ le_1 := exp_1[\textbf{self}.x],\ \textbf{self}.x := exp_2]$
provided
($\rightarrow$) $getX$ is not declared in any superclass or subclass of A in cds;
$setX$ is not declared in any superclass or subclass of A in cds;
($\leftarrow$) $le.getX$ and $le.setX$ do not appear in mts'_a, cds or c, for any le such that $le \leq A$.

3.3 Move Method

Moving methods between classes helps to make them simpler and clearer, improving legibility. Rule 3 $\langle Move\ Method \rangle$ allows us to move a method that uses more features of another class than the class on which it is defined.

On the left-hand side of this rule, method m_1 of A reads and writes to the attribute x of B by means of its get and set methods. The method calls occur inside the command c_1 of the method m_1 and have as target the attribute b. The variable aux holds the result of the call to the method $getX$ of B. Such variable may occur on the right-hand side of an assignment to an expression le. In the command c_1, there may also be occurrences of the parameters that appear in pds, which is indicated by $\alpha(pds)$, where $\alpha(pds)$ gives the list of parameter names declared in pds. For instance, if pds is the declaration $(\textbf{val}\ x : T; \textbf{res}\ y : T)$, the list given by $\alpha(pds)$ is $x; y$. The class B on the left-hand side declares no method like m_1 of the class A.

On the right-hand side of this rule, the method m_1 in A becomes just a delegating method: it calls the corresponding method of class B and passes the parameters in pds as arguments. The class B declares a method called m_1 similar to the method m_1 that appears on the left-hand side of the rule. The attribute x is accessed by means of the get and set methods declared in B itself.

To apply this rule, we assume that $b \neq \textbf{null}$ and $b \neq \textbf{error}$ along the lifetime of an object of type A, otherwise a call to the method m_1 of the class A would abort because this method calls methods of the class B. Also, the parameters arg_1 and arg_2 must have basic types.

In order to apply Rule 3 $\langle Move\ Method \rangle$ from left to right, the method m_1 must not be declared in B, nor in any of its superclasses or subclasses, and there should be no occurrences of **super** in m_1. Also, attributes declared in A are not accessed in m_1, and methods declared in A are not called inside m_1. The reason for the last two provisos is that, to move a method in which there are occurrences of attributes or calls to methods of the source class (the one that introduces the method), it would be necessary to pass self as argument in the calls to the method of the target class

Rule 3 ⟨*Move Method*⟩

class A **extends** C
  pri b : B ; ads_a
  meth $m_1 \mathrel{\widehat{=}} (pds \bullet$
    c_1[**var** aux : $T\bullet$
    self.$b.getX(aux)$;
    $le := exp_1[aux]$ **end**,
    self.$b.setX(exp_2)$, $\alpha(pds)$]))
  new $\mathrel{\widehat{=}}$ (**self**.$b := new\ B()$)
  mts_a
end
class B **extends** D
  pri x : T ; ads_b
  meth $getX \mathrel{\widehat{=}}$
    (**res** arg : $T\bullet$ arg := **self**.x)
  meth $setX \mathrel{\widehat{=}}$
    (**val** arg : $T\bullet$ **self**.$x := arg$)
  mts_b
end

$=_{cds,c}$

class A **extends** C
  pri b : B ; ads_a
  meth $m_1 \mathrel{\widehat{=}} (pds \bullet$
    self.$b.m_1(\alpha(pds))$)
  new $\mathrel{\widehat{=}}$ (**self**.$b := new\ B()$)
  mts_a
end
class B **extends** D
  pri x : T ; ads_b
  meth $getX \mathrel{\widehat{=}}$
    (**res** arg : $T\bullet$ arg := **self**.x)
  meth $setX \mathrel{\widehat{=}}$
    (**val** arg : $T\bullet$ **self**.$x := arg$)
  meth $m_1 \mathrel{\widehat{=}}$
    ($pds \bullet c_1$[**var** aux : $T\bullet$
    self.$getX(aux)$;
    $le := exp_1[aux]$ **end**,
    self.$setX(exp_2)$, $\alpha(pds)$]))
  mts_b
end

where
$\alpha(pds)$ gives the names of parameters in pds;
provided
($\leftrightarrow$) $b \neq$ **null** $\wedge\ b \neq$ **error** is an invariant of A; arg_1 and arg_2 have basic types;
($\rightarrow$) m_1 is not declared in mts_b nor in any superclass or subclass of B; **super** does not appear in m_1; **self**.a does not appear in m_1, for any a declared in ads_a; **self**.p does not appear in m_1, for any method p in mts_a;
($\leftarrow$) $N.m_1$ does not appear in mts_b, cds or c, for any N such that $N \leq B$.

(the class to where we move the method). In this way, the parameter declaration of the method in the target class would include an additional parameter: one whose type is the source class. Also, clients of the target class that call the method that was moved would also need to declare an object of the source class and pass such an object as argument in the method call. To avoid the additional parameter, we restrict the method m_1 not to access attributes nor call methods declared in A. For applying this rule from right to left, the method m_1 must not be called in mts_b, cds, or c, as this method is removed from B.

4 CafeOBJ

CafeOBJ [18] is a new generation algebraic specification and programming language. As a successor of the OBJ family (OBJ1, OBJ2, OBJ3) [11], it inherits features such as: powerful typing system with sub-types; sophisticated module composition system, featuring several kinds of imports; parameterised modules; views for instantiating parameters and the module expressions, among other issues. CafeOBJ implements new paradigms, such as rewriting logic and hidden algebra, as well as their combinations. It is mainly used for system specification, formal verification of specifications, rapid prototyping, programming and automatic theorem proving.

CafeOBJ is chosen due to some characteristics, among them, the availability of documentation, the facility in the use of the reduction mechanism, the possibility of applying the rules in two ways and in subterms of the term to be reduced.

A module in CafeOBJ has the syntax defined by `module < mod_id > mod_elem*`, where `< mod id >` is the name of the module and `mod_elem` is an element of the module. A module element is either an import declaration, a sort declaration, an operator declaration, a record declaration, a variable declaration, an equation

```
module PEANO-NAT {
  imports {
    protecting (NAT)
    protecting (INT)
  }
  signature {
    [ Peano-Nat , Nat < Int ]
    op 0 : -> Peano-Nat
    op s : Peano-Nat -> Peano-Nat
    op (_+_) : Peano-Nat Peano-Nat -> Peano-Nat
  }
  axioms {
    var N : Peano-Nat
    eq 0 + N = N .
    eq s(M:Peano-Nat) + N = s(M + N) .
  }
}
```

Figure 1. The module PEANO-NAT in CafeOBJ.

declaration or a transition declaration. These elements are structured into three main parts. The first part, imports, specifies which modules should be imported, that is, inherited. There are three forms of importing modules: `protecting` (the imported module can not be changed), `extending` (the imported module can be extended, but the original description remains unchanged) and `using` (the imported module can be extended or can change the original description). The second part, signature, declares sorts, operators, records and subsorts used by the module. Finally, axioms includes declaration of variables, equations and transitions and expresses the behavior of the module.

To illustrate a module description in CafeOBJ, consider the example of Figure 1, which defines the Peano notation for natural numbers. The module `PEANO-NAT` inherits the sorts and the operators defined in modules `INT` and `NAT`. Section `signature` declares the sorts `Peano-Nat`, `Nat` and `Int`. The symbol `<` means that `Nat` is a subsort of `Int`. The zero constant, the `s` and the (infix) `+` operators are introduced by `op`. The behaviour of the `+` operator is given by two expressions, introduced by `eq`.

5 Proving and Implementing Refactoring Rules in CafeOBJ

The mechanisation of the proofs of refactoring rules comprises two steps: the implementation of the ROOL grammar and laws (Section 5.1) and the automatic derivation of the rules from the laws already implemented (Section 5.2). After proved, the refactoring can be implemented as a rule and used to perform program transformation (sections 5.3 and 5.4).

5.1 The Implementation of ROOL *Grammar and Laws*

To implement the grammar, a module `ROOL-GRAMMAR` is defined, including the language operators and constructors. Every expression or command in ROOL is thus defined by using these operators and constructors. The full implementation of the grammar comprises 92 operators and a small fragment of this module is depicted

in Figure 2, where the lines are numbered for didactic reasons. This facility is used in the remainder of this paper.

```
 1  module ROOL-GRAMMAR {
 2      [ ClassName PrimitiveType < Type ]
 3      op bool    : -> PrimitiveType .
 4      op int     : -> PrimitiveType .
 5      op char    : -> PrimitiveType .
 6      ...
 7      op _is_    : Expression ClassName      -> Bool .
 8      op (_)_    : ClassName Expression      -> Expression .
 9      op _._     : Expression Variable       -> Expression .
10      ...
11      op _,_     : Variable VariableList  -> VariableList { r-assoc } .
12      ...
13      op _->_    : Bool Command -> GuardedCommand { prec: 39 l-assoc } .
14      op _[]_    : GuardedCommandList GuardedCommandList ->
             GuardedCommandList { ... } .
15      ...
16      op (if_fi) : GuardedCommandList -> Command .
17      ...
18      op (val_:_)  : VariableList TypeList -> ParDecl .
19      ...
20      op class_extends__end : ClassName ClassName ClassBody ->
             ClassDecl {prec: 40} .
21      op class__end         : ClassName ClassBody           ->
             ClassDecl {prec: 40} .
22      ...
23      op (pri_:_;)  : Variable Type  -> Attr { prec: 0 } .
24      ...
25      op (meth_^=_) : MethodName ParCommand -> Meth { prec: 0 } .
26      ...
27  }
```

Figure 2. A fragment of the module ROOL-GRAMMAR in CafeOBJ.

The module is introduced by the reserved word `module`. In Line 2 the sorts `ClassName` and `PrimitiveType` are declared as subsorts of `Type`. Lines 3 to 5 define some primitive types of ROOL. The type test, cast and the attribute selection operators are defined in lines 7 to 9, respectively. For example, the type test operator receives an expression and a name of a class, and returns true whenever the type of the expressions is equal to the given class name. After that, some auxiliary operators are defined, such as the one in Line 11, which creates lists of variables. As an example of command implementation, the guarded command is depicted in lines 13 and 14, and the alternation in Line 16. Parameters declaration is implemented as in Line 18, which shows the call-by-value mechanism. Class declarations, considering superclasses or not, are shown in lines 20 and 21, respectively. In Line 20 the second `ClassName` corresponds to the superclass. Private attributes are declared using the operator in Line 23 and Line 25 introduces a method declaration. The reserved word `prec` introduces the precedence of the operator.

To implement the laws, a module `ROOL-LAWS` is defined, which imports modules `ROOL-UTILS` and `ROOL-LEMMAS`. The former imported module includes some auxiliary operators, like the constructors for integer lists, necessary to implement the laws, whereas the latter one includes some partial results, used to derive the refactoring rules. Each law is defined as a equation in CafeOBJ. The module `ROOL-LAWS` comprises the implementation of seventy laws, the module `ROOL-UTILS` the implementation of seven auxiliary operators and the module `ROOL-LEMMAS` the implementation of six intermediary results, including lemmas 1 and 2 of Section 3.

```
 1  module ROOL-LAWS {
 2      protecting (ROOL-LEMMAS)
 3      protecting (ROOL-UTILS)
 4      ...
 5      var x : Variable .
 6      var T : Type .
 7      var c : Command .
 8      op replaceAndAppend : GuardedCommandList LeftExp Expression ->
           GuardedCommandList .
 9      eq replaceAndAppend (p -> c, le, e) = (p [ e / le ]) -> (le := e ; c)
           .
10      eq replaceAndAppend ((p -> c) [] gcl , le, e) =
11                          (p [ e / le ]) -> (le := e ; c) []
                               replaceAndAppend(gcl, le, e) .
12      eq [:=-<>-right-dist] : le := e ; if gcl fi = if replaceAndAppend(gcl
           , le, e) fi .
13      ...
14      eq [var-elim] : var x : T @ c end = c .
15      ...
16      eq [var-:=-final-value] : var x : T @ c ; x := e end = var x : T @ c
           end .
17      ...
18      trans [var-:=initial-value] : var x : T @ c end => var x : T @ x := e
            ; c end .
19      ...
20      eq [order-independent-assignment] : x := e1 ; y := e2 = y := e2 ; x
           := e1 .
21      ...
22      eq [method-elimination] :
23          class C extends D ads meth m ^= pc ops end =
24          class C extends D ads              ops end .
25      eq [move-original-method-to-superclass] :
26          class B extends A ads    ops end class C extends B ads' meth m ^=
                pc ops' end =
27          class B extends A ads meth m ^= pc ops end class C extends B ads'
                  ops' end .
28      ...
29  }
```

Figure 3. A fragment of the module ROOL-LAWS in CafeOBJ.

To illustrate this module, Figure 3 shows a small fragment of the generated code. Basically some variables used in the laws equations are introduced, as the ones in Lines 5 to 7. After that, the command laws are implemented. Lines 8 through 12 depicts the implementation of Law 1 $\langle := - \lhd \rhd \mathit{right\ dist} \rangle$, given in Section 2. An extra operator, `replaceAndAppend`, which is responsible for inserting the substitution and moving the assignment, is necessary, as the law is applied in a list of guarded commands with an arbitrary number of elements and therefore cannot be directly implemented. Lines 14, 16 and 20 show, respectively, the implementation of Law 2 $\langle \mathit{var\ elim} \rangle$, 3 $\langle \mathit{var}\text{-} := \ \mathit{final\ value} \rangle$ and, 5 $\langle \mathit{order\ independent\ assignment} \rangle$, explained in Section 2. The implementation of these laws is immediate where the symbol "`@`" means "•" in the formal definition. Law 4 $\langle \mathit{var}\text{-} := \ \mathit{initial\ value} \rangle$ is implemented in Line 18, as a transition by using the reserved word `trans`, as Law 4 is a refinement and not an equality. The laws of classes are also implemented in this module. For example, lines 22 to 24 introduce Law 6 $\langle \mathit{method\ elimination} \rangle$, whereas lines 25 to 27 define Law 7 $\langle \mathit{move\ original\ method\ to\ superclass} \rangle$. The implementation of both laws follows directly from the formal definition presented in Section 2. For now we have not implemented the verification of the conditions that must be satisfied to apply some laws and we suppose that they are true.

5.2 Proving a Refactoring Rule

In this section we present the derivation of the Extract/Inline Method refactoring, using CafeOBJ. The manual proof of this refactoring, given by Cornélio in [8], is reproduced in Section 3.

```
start class A:ClassName extends C:ClassName ads:AttrS meth m:MethodName ^= (pds1:ParDecls
@ c1:Command [ c2:Command [ a:Variable ] ]) mts:MethS end .


apply -.method-elimination with m = m2:MethodName, pc = (val arg:Variable :  T:Type @
c2:Command [ arg ] ) within term .

result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName ^= (( val
arg:Variable :  T:Type) @ (c2:Command [ arg:Variable ])) ( meth m:MethodName ^=
(pds1:ParDecls @ (c1:Command [ (c2:Command [ a:Variable ]) ])) mts:MethS) end :  ClassDecl


apply .lemma2 with vl = arg, x = a:Variable at (3 2 2 1 2 2 2) .

result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName ^= (( val
arg:Variable :  T:Type) @ (c2:Command [ arg:Variable ])) ( meth m:MethodName ^=
(pds1:ParDecls @ (c1:Command [ (((val arg:Variable :  T:Type) @ (c2:Command [ arg:Variable
])) < a:Variable >) ])) mts:MethS) end :  ClassDecl


apply .lemma1 with m = m2 at (3 2 2 1 2 2 2) .

result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName ^= (( val
arg:Variable :  T:Type) @ (c2:Command [ arg:Variable ])) ( meth m:MethodName ^=
(pds1:ParDecls @ (c1:Command [ (self.  m2:MethodName < a:Variable >) ])) mts:MethS) end
:  ClassDecl
```

Figure 4. Automatic derivation of Rule 1 ⟨Extract/Inline Method⟩.

Figure 4 shows the automatic derivation of that refactoring. Firstly, the left-hand side of the rule is introduced by the command `start`. After that, each law and lemma used in the derivation process is applied, using the command `apply`. The results achieved after each application follows the reserved word `result`. By comparing this figure with the proof given in Section 3, observe that the first result shows the transformation of the left-hand side of the rule after applying Law 6 ⟨*method elimination*⟩. To perform the application of this law the generic variables `m` and `pc` are instantiated with the corresponding terms on the left-hand side of the refactoring. The result achieved is very similar with the one of Section 2. The major difference is that each variable is associated with its type. After that, lemmas 1 and 2 of module `ROOL-LEMMAS` reduce this partial result to the right-hand side. The numbers that appear after the reserved word `at` are parameters used to capture the term in which the transformation is applied. For example, Lemma 2 is applied to transform the term `c2[a]` inside the body of method `m`. The rest of the refactoring rules proposed in [8] are reduced in a similar way.

5.3 Implementation and Application of Rule 2 < Self Encapsulate Field >

After proving the refactoring rules, they can be implemented as equations in CafeOBJ. To implement Rule 2 ⟨ *Self Encapsulate Field* ⟩ in CafeOBJ, a module `ROOL-SELFENCAPSULATE` is defined, whose fragment is depicted in Figure 5.

This module imports all operators from `ROOL-GRAMMAR`. Lines 4 to 11 define all operators necessary to implement this rule. `SelfEncapsulate` is the main operator, which receives the class declaration to be refactored, the attribute and the names

```
 1  module ROOL-SELFENCAPSULATE {
 2      protecting (ROOL-GRAMMAR)
 3      ...
 4      op SelfEncapsulate          : ClassDecl Variable MethodName
            MethodName -> ClassDecl .
 5      op getType                  : AttrS Variable -> Type .
 6      op replaceOnMethod          : MethS Variable MethodName MethodName
            Type -> MethS .
 7      ops replaceOnCommand replaceOnAssignmentList replaceOnAssignment :
 8              Command Variable MethodName MethodName Type -> Command .
 9      op replaceOnExpression      : Expression Variable -> Expression .
10      op containsAttribute        : Expression Variable -> Bool .
11      ops fromListToConcat fromConcatToList : Command -> Command .
12      ...
13      eq SelfEncapsulate( class A extends B as ms end , x , getX , setX )
             =
14          class A extends B
15              as
16              (meth getX ^= ( (res x : getType(as, x)) @ (x := self. x) )
                    )
17              (meth setX ^= ( (val x : getType(as, x)) @ (self. x := x) )
                    )
18              replaceOnMethod(ms, x, getX, setX, getType(as, x))
19          end .
20      ...
21      eq getType( (pri  a : tipo:Type ;) as , a ) = tipo .
22      ...
23      eq replaceOnMethod( (meth m ^= (pd @ c)) ms, x, getX, setX, T) =
24          (meth m ^= ( pd @ replaceOnCommand(c, x, getX, setX, T) ))
25          replaceOnMethod(ms, x, getX, setX, T) .
26      ...
27      eq replaceOnCommand( leL := eL , x, getX, setX, T) =
28          fromConcatToList(replaceOnAssignmentList(
29                          fromListToConcat(leL := eL), x, getX, setX, T) )
                                .
30      ...
31      eq replaceOnAssignmentList( le := e ; c , x, getX, setX, T) =
32         replaceOnAssignment( le := e , x, getX, setX, T)
33                          ; replaceOnAssignmentList( c , x, getX, setX, T)
                                .
34      ...
35      ceq replaceOnAssignment( le := e , x, getX, setX, T) =
36         var x : T @ self. getX < x > end ; le := replaceOnExpression(e,
               x)
37                                                 if containsAttribute(e, x)
                                                        .
38      ...
39      eq replaceOnAssignment(self. x := e,x,getX,setX,T)=self. setX < e >
            .
40      ...
41      eq replaceOnExpression( e <= e2 , x ) = replaceOnExpression(e, x)
            <=
42                                                replaceOnExpression(e2, x)
                                                      .
43      ...
44      eq replaceOnExpression( self. x , x ) = x .
45      eq replaceOnExpression( e , x ) = e .
46      ...
47  }
```

Figure 5. A fragment of the module ROOL-SELFENCAPSULATE in CafeOBJ.

of the newly introduced get and set methods. The `getType` operator simply returns the type of the given attribute present in the list of attributes `as`. Operators `fromListToConcat` and `fromConcatToList` make the term simpler for the applications of other operations. Basically, they turn an assignment of a list of variables to a list of expressions into a concatenation of assignments, with each variable assigned to its corresponding expression in the list. The function `containsAttribute` receives an expression and an attribute name and returns true if such attribute occurs inside the given expression. The function `replaceOnMethod` replaces inside the

given methods all the occurrences of the given attribute by its get or set methods. All the remaining operators behave in a similar way.

```
1 start SelfEncapsulate( SelfEncapsulate( class IntRange pri _low : int
      ; pri _high : int ; meth includes ^= ( val arg : int ; res ret :
      bool @ ( ret := ((arg >= self. _low) && (arg <= self. _high)) ) )
      meth grow ^= ( val factor : int @ ( self. _high := self. _high *
      factor ) ) end , _high, getHigh , setHigh ) , _low, getLow ,
      setLow ) .
2 apply red at term .
3 result class IntRange:ClassName pri _low:Variable : int ; pri _high:
      Variable : int ; meth getLow:MethodName ^= (( res _low:Variable :
      int) @ (_low := self. _low)) (meth setLow:MethodName ^= (( val
      _low : int) @ ( self. _low := _low)) (meth getHigh:MethodName ^=
      (( res _high:Variable : int) @ (_high := self. _high)) (meth
      setHigh:MethodName ^= (( val _high : int) @ ( self. _high := _high
      )) (meth includes:MethodName ^= ((( val arg:Variable : int);( res
      ret:Variable : bool))@ var _high : int @ ( self. getHigh < _high >
       ; var _low : int @ ( self. getLow < _low > ; ret:Variable := (arg
      :Variable >= _low) && (arg:Variable <= _high)) end) end) meth grow
      :MethodName ^= (( val factor:Variable : int) @ var _high : int @ (
       self. getHigh < _high > ; self. setHigh < (_high * factor:
      Variable) >) end))))) end : ClassDecl
```

Figure 6. Application of the implementation of Rule 2 ⟨Self Encapsulate Field⟩.

```
class IntRange
    pri _low : int;
    pri _high : int;
    meth includes ≙ (val arg : int; res ret : bool •
                ret := (arg >= self._low) && (arg <= self._high))
    meth grow ≙ (val factor : int • self._high := self._high * factor)
end
```

Figure 7. Class *IntRange*, used in the application.

The effective application of the implemented rule in an arbitrary example, considering that the provisos are satisfied, is shown in Figure 6. The example introduced by the `start` command is extracted from Fowler [10], and reproduced in Figure 7. The class declaration must be supplied to CafeOBJ inside the operator `SelfEncapsulate`, along with the name of the attribute to be encapsulated, as well as the names of the get and set methods to be created. Observe that in the transformed program all assignments to the encapsulated attributes use set methods. Moreover, where these attributes are only used, a get method is introduced.

5.4 Implementation and Application of Rule 3 < Move Method >

Rule 3 ⟨*Move Method*⟩ is used for moving methods between classes, helping to make them simpler and clearer, improving legibility.

Figure 8 shows a fragment of the module `ROOL-MOVEMETHOD`. Like the previous implementation, the module `ROOL-GRAMMAR` is imported. Firstly, some variables to be used in the equations are defined, lines 3 to 7. Lines 9 to 18 show the declaration of the most important operators of this implementation. The main operator, `MoveMethod`, has its behaviour defined by two conditional equations. One of them is shown in lines 20 to 23, the other one is similar. Basically, this operators assigns the class declaration where the method to be moved is (source class) and the class to where it should be moved (target class). Lines 39 to 42 define the

```
 1  module ROOL-MOVEMETHOD {
 2      protecting (ROOL-GRAMMAR)
 3      var bd : ClassBody .
 4      vars mm m : MethodName .
 5      vars Co Ct : ClassName .
 6      vars cdl cdl1 cdl2 : ClassDecl .
 7      var pard : ParDecl .
 8      ...
 9      op MoveMethod : ClassDecls ClassName ClassName MethodName ->
            ClassDecls .
10      op usesAttributeOfTargetClass : ClassDecl MethodName -> Bool .
11      ...
12      op removeMethod : ClassDecl Bool ClassName MethodName -> ClassDecl
            .
13      op createDelegate : MethS MethodName LeftExp -> MethS .
14      op deleteMethod : MethS MethodName -> MethS .
15      ...
16      op createMethod : ClassDecl Bool ClassDecl MethodName -> ClassDecl
            .
17      op getMethodPC : ClassDecl MethodName -> ParCommand .
18      op getClassName : ClassDecl -> ClassName .
19      ...
20      ceq MoveMethod( cdl1 cdl2 , Co, Ct, m ) =
21           removeMethod(cdl1, usesAttributeOfTargetClass(cdl1,m), Ct, m )
22           createMethod(cdl2, usesAttributeOfTargetClass(cdl1,m), cdl1, m
                 )
23           if (getClassName(cdl1) == Co) and (getClassName(cdl2) == Ct).
24      ...
25      eq removeMethod(class A extends B as ms end, false, Ct, m) =
26              class A extends B as deleteMethod(ms, m) end .
27      eq removeMethod(class A extends B as ms end,  true, Ct, m) =
28              class A extends B as createDelegate(ms, m, self. getCTattr(
                    as,Ct) ) end .
29      ...
30      eq createDelegate( (meth mm ^= (pd @ c)) ms, m , le) =
31      (meth mm ^= (pd @ c)) createDelegate(ms, m, le) .
32      eq createDelegate( (meth mm ^= (pd @ c))   , mm, le) =
33                               (meth mm ^= (pd @ (le . mm < alpha(pd) >)))
                                     .
34      ...
35      eq deleteMethod( (meth mm ^= (   @ c)) ms, m ) =
36                                (meth mm ^= (   @ c)) deleteMethod(ms, m)
                                     .
37      eq deleteMethod( (meth mm ^= (pd @ c)) ms, mm) = ms .
38      ...
39      eq createMethod(class A extends B as ms end, false, cdl, m) =
40           class A extends B as ((meth m ^= getMethodPC(cdl, m)) ms) end
                .
41      eq createMethod(class A extends B as ms end, true, cdl, m) =
42           class A extends B as ((meth m ^= replaceSelfs(getMethodPC(cdl,
                m))) ms) end .
43      ...
44      eq getMethodPC(class A extends B as ms end, m) = getMethodPC2(ms,m)
            .
45      eq getClassName(class A extends B as ms end) = A .
46  }
```

Figure 8. O módulo `ROOL-MOVEMETHOD` em CafeOBJ.

`createMethod` operator, who creates in the target class the new method. The operator `removeMethod` is called upon the source class and deletes the method (through the operator `deleteMethod`, lines 35 to 37) or turns it into a delegation, using the function `createDelegate`, lines 30 to 33. All the remaining operators are auxiliary.

Considering the example of Figure 9, the application of this implementation is shown in Figure 10. We begin by introducing the classes using the `start` command, inside the `MoveMethod` operator. The `print` command, Line 1, denotes a built-in function of ROOL. The parameters of the operator `MoveMethod` are the class where the method to be moved is, the class to where the method should be moved and the

```
class House
   pri _owner : User;
   meth printInfo ≙ (val header : string • print(header);
      var name : string • self._owner.getName(name); print(name) end;
      print(self._owner._childrenCount))
   meth anotherMethod ≙ ( • c)
end
class User
   pri _name : string; pub _childrenCount : int;
   meth getName ≙ (res n : String • n := self._name )
end
```

Figure 9. Initial *House* and *User* classes.

```
1 start MoveMethod( class House pri _owner : User ; meth printInfo ^= (val
      header : String @ ( print < header > ; var name : String @ (self.
     _owner . getName < name >) ; print < name > end ; print < (self.
     _owner . _childrenCount) > ) ) meth anotherMethod ^= ( @ c ) end
     class User pri _name : String ; pub _childrenCount : int ; meth
     getName ^= ((res n : String) @ (n := self. _name)) end , House, User
     , printInfo ) .
2 apply red at term .
3 result class House:ClassName pri _owner:Variable : User:ClassName ; meth
      printInfo:MethodName ^= ((val header:Variable : String:ClassName) @
      ((self. _owner . printInfo) < header >)) meth anotherMethod:
     MethodName ^= (@ c:Command) end class User:ClassName pri _name:
     Variable : String:ClassName ; pub _childrenCount:Variable : int:Type
      ; meth printInfo ^= ((val header : String) @ (print:ParCommand <
     header:Variable > ; (var name:Variable : String:ClassName @ (self.
     getName:MethodName < name:Variable > ; print:ParCommand < name:
     Variable >) end ; print:ParCommand < (self. _owner:Variable .
     _childrenCount :Variable) >))) meth getName:MethodName ^= ((res n:
     Variable : String:ClassName) @ (n:Variable := self. _name:Variable))
      end : ClassDecls
```

Figure 10. Application of Rule 3 ⟨*Move Method*⟩.

name of such method. In the example of Figure 10, `House` is the source class, `User` is the target class and `printInfo` is the method to be moved. Observe that in the transformed program the `printInfo` method in the `House` class is now a delegation to the `printInfo` method created in the `User` class.

6 Case Study

In order to illustrate the application of the implemented refactoring rules, a brief case study was developed: a program that maintains information about the products, suppliers and sales of a store.

A class diagram of the initial design, described using UML (*Unified Modeling Language*) [13], is depicted in Figure 11, where we can see several classes, representing the various concepts of the program. In the context of this case study, the classes *Client*, *Supplier*, *Brand*, *Sale* and *Salesman* are merely illustrative. Therefore, their implementations are not relevant and we consider that all their attributes are public. The remaining classes – *Product* and *Item* – will be transformed by applying refactorings. Their initial declaration are shown in Figure 12, where *print* is a built-in function of ROOL.

In this design, the class *Product* only defines something that can be sold, it does not include price or quantity. To deal with the price and quantity, there is a class called *Item*, which has some properties like *acquisitionDate*, *purchasePrice*, *profit*,

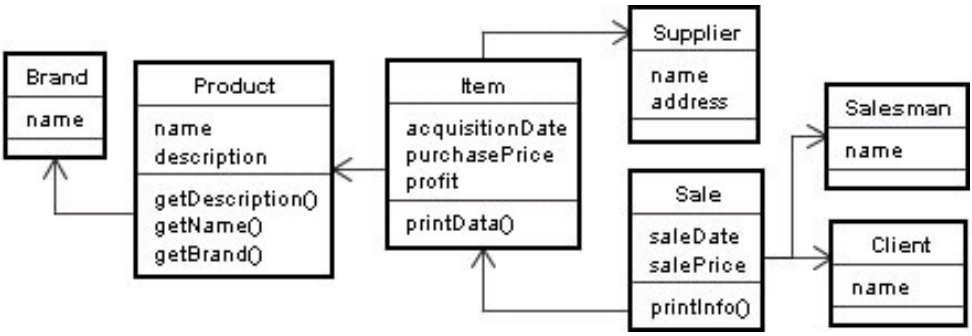


Figure 11. Design of a product and sales control program.

```
class Item
      pri acquisitionDate : string; pri purchasePrice : double;
      pri profit : double; pri supp : Supplier; pri prod : Product;
      meth printData ≙ (•print(self.acquisitionDate); print(self.purchasePrice);
            print(self.profit); print(self.supp.name);
            var n : string • self.prod.getName(n); print(n) end;
            var d : string • self.prod.getDescription(d); print(d) end;
            var m : Brand • self.prod.getBrand(m); print(m.name) end)
end
class Product
      pri name : string; pri brand : Brand; pri description : string;
      meth getName ≙ (res r : string • r := self.name)
      meth getBrand ≙ (res r : Brand • r := self.brand)
      meth getDescription ≙ (res r : string • r := self.description)
end
```

Figure 12. Classes *Item* and *Product* in ROOL.

prod e *supp*, which represent, respectively, the date when the product was acquired from the supplier, the price payed to the supplier, the value (percent) to be added to that price, the product itself and the supplier of this specific item. To represent the sale of an item, the class *Sale* is used. Among other attributes, this class maintains the sold item, who sold it and to whom it was sold. Therefore, when, for example, one wants to know if there are any products for sale it is possible to look for objects of *Item* that are not sold.

One of the first problems to be noticed in the code of Figure 12 is the size of the *printData* method in the *Item* class. Observe that its commands can be divided into two distinct logic blocks: the first two lines print information about the item itself, whereas the other ones deal with the product only. In order to obtain a clearer and more readable method definition, it is possible to extract the last three lines in to a separate command. To achieve this, Rule 1 ⟨*Extract Method*⟩ is used and a new method called *printProd* is created.

Figure 13 depicts how such refactoring is applied to the *Item* class. The first step is to introduce, through the `start` command, the class to be refactored, in this case the class `Item`. After that, we use two auxiliary equations: `markCommandToBeExtracted` and `introduceExtractMethod`, which assign, respectively, the list of commands to be extracted and the class in which the extracted commands are replaced by a call to the new method. These equations are used so that no manual treatment of the code is necessary before introducing it to the rewriting system. The last step is to request the reduction of the current term, in

```
start class Item pri acquisitionDate : string ; pri purchasePrice : double
    ; pri profit : double ; pri supp : Supplier ; pri prod : Product ; meth
     printData ^= ( @ (print < self. acquisitionDate > ; print < self.
    purchasePrice > ; print < self. profit > ; print < self. supp . name >
    ; var n : string @ (self. prod . getName < n > ; print < n >) end ; var
     d : string @ (self. prod . getDescription < d > ; print < d >) end ;
    var m : Brand  @ (self. prod . getBrand < m > ; print < m . name >) end
    ) ) end .
apply .markCommandToBeExtracted at (2 2 2 1 2 2 2 2) .
apply .introduceExtractMethod with methodname = printProd at term .
apply red at term .
result class Item:ClassName pri acquisitionDate:Variable : string ; (pri
    purchasePrice:Variable : double ; ( pri profit:Variable : double ; (
    pri supp:Variable : Supplier:ClassName ; pri prod:Variable : Product:
    ClassName ;))) meth printProd:MethodName ^= ( @ (var n:Variable :
    string @ ((self. prod:Variable . getName:MethodName) < n:Variable > ;
    print:ParCommand < n:Variable >) end ; (var d:Variable : string @ ((
    self. prod:Variable . getDescription:MethodName) < d:Variable > ; print
    :ParCommand < d:Variable >) end ; var m:Variable : Brand:ClassName @ ((
    self. prod:Variable . getBrand:MethodName) < m:Variable > ; print:
    ParCommand < (m:Variable . name:Variable) >) end))) meth printData:
    MethodName ^= (@ (print:ParCommand < self. acquisitionDate:Variable > ;
     (print:ParCommand < self. purchasePrice:Variable > ; (print:ParCommand
     < self. profit:Variable > ; (print:ParCommand < ( self. supp:Variable
    . name:Variable) > ; self. printProd < >))))) end : ClassDecl
```

Figure 13. Extracting the method *printProd* using Rule 1 ⟨*Extract Method*⟩.

this case, the class to be refactored.

class *Item*
  pri *acquisitionDate* : **string**; **pri** *purchasePrice* : **double**;
  pri *profit* : **double**; **pri** *supp* : *Supplier*; **pri** *prod* : *Product*;
  meth *printProd* ≙ (• **var** *n* : **string** • **self**.*prod*.*getName*(*n*); *print*(*n*) **end**;
    var *d* : **string** • **self**.*prod*.*getDescription*(*d*); *print*(*d*) **end**;
    var *m* : *Brand* • **self**.*prod*.*getBrand*(*m*); *print*(*m.name*) **end**)
  meth *printData* ≙ (• *print*(**self**.*acquisitionDate*); *print*(**self**.*purchasePrice*);
    print(**self**.*profit*); *print*(**self**.*supp.name*);
    self.*printProd*())
end

Figure 14. Classe *Item* after extraction of method *printProd*.

The *Item* class obtained after the refactoring is shown, described in ROOL, in Figure 14, for didactic purposes. Observe that it now has a method called *printProd* which deals only with the attribute *prod* of type *Product*. Notice that the method *printData* now has a call to the newly introduced method *printProd*.

In order to reduce the coupling between the classes *Item* and *Product*, the method *printProd* can be moved to the class *Product*. To do so, Rule 3 ⟨*Move Method*⟩ is applied, as shown if Figure 15. Similarly to the previous application, it is necessary to use an auxiliary equation, `introduceMoveMethod`, to specify which classes will be refactored. The *Product* and *Item* classes after the refactoring are shown in Figure 16, for didactic reasons. Notice that the method *printProd* now belongs to the class *Product*. The final class diagram is depicted in Figure 17.

7 Conclusions

This paper illustrates how rewriting systems, in particular CafeOBJ, can be used to mechanise refactoring proofs, in the algebraic style developed by Cornélio in [8]. In this way, this work complements the formal rigour of that approach. Moreover, the grammar of ROOL as well as its programming laws are fully implemented. Nev-

```
start class Product pri name : string ; pri brand : Brand ; pri description
    : string ; meth getName ^= (res r : string @ (r := self. name)) meth
    getBrand ^= (res r : Brand @ (r := self. brand)) meth getDescription ^=
     (res r : string @ (r := self. description)) end class Item pri
    acquisitionDate : string ; pri purchasePrice : double ; pri profit :
    double ; pri supp : Supplier ; pri prod : Product ; meth printProd ^= (
     @ ( var n : string @ ((self. prod . getName) < n > ; print < n >) end
    ; (var d : string @ ((self. prod . getDescription) < d > ; print < d >)
     end ; var m : Brand @ ((self. prod . getBrand) < m > ; print < (m .
    name) >) end))) meth printData ^= (@ (print < self. acquisitionDate > ;
     (print < self. purchasePrice > ; (print < self. profit > ; (print < (
    self. supp . name) > ; self. printProd < >))))) end .
apply .introduceMoveMethod with Co = Item:ClassName, Ct = Product:ClassName
    , methodname = printProd:MethodName at term .
apply red at term .
result class Item:ClassName pri acquisitionDate:Variable : string ; (pri
    purchasePrice:Variable : double ; ( pri profit:Variable : double ; (
    pri supp:Variable : Supplier:ClassName ; pri prod:Variable : Product:
    ClassName ;))) meth printProd:MethodName ^= ( @ (( self. prod .
    printProd) < >)) meth printData:MethodName ^= (@ ( print:ParCommand <
    self. acquisitionDate:Variable > ; (print:ParCommand < self.
    purchasePrice:Variable > ; (print:ParCommand < self. profit:Variable >
    ; (print:ParCommand < ( self. supp:Variable . name:Variable) > ; self.
    printProd:MethodName < >))))) end class Product:ClassName pri name:
    Variable : string ; (pri brand:Variable : Brand:ClassName ; pri
    description:Variable : string ;) meth printProd ^= ( @ ( var n:Variable
     : string @ ( self. getName:MethodName < n:Variable > ; print:
    ParCommand < n:Variable >) end ; ( var d:Variable : string @ ( self.
    getDescription:MethodName < d:Variable > ; print:ParCommand < d:
    Variable >)  end ; var m:Variable : Brand:ClassName @ ( self. getBrand:
    MethodName < m:Variable > ; print:ParCommand < (m:Variable . name:
    Variable) >) end))) (meth getName:MethodName ^= ((res r:Variable :
    string) @ (r:Variable := self. name:Variable)) (meth getBrand:
    MethodName ^= ((res r:Variable : Brand:ClassName) @ (r:Variable := self
    . brand:Variable)) meth getDescription:MethodName ^= ((res r:Variable :
     string) @ (r:Variable := self. description:Variable)))) end :
    ClassDecls
```

Figure 15. Application of Rule 3 ⟨*Move Method*⟩ in *Product* and *Item*.

```
class Product
      pri name : string; pri brand : Brand; pri description : string;
      meth printProd ≙ (• var n : string • self.getName(n); print(n) end;
            var d : string • self.getDescription(d); print(d) end;
            var m : Brand • self.getBrand(m); print(m.name) end)
      meth getName ≙ (res r : string • r := self.name)
      meth getBrand ≙ (res r : Brand • r := self.brand)
      meth getDescription ≙ (res r : string • r := self.description)
end
class Item
      pri acquisitionDate : string; pri purchasePrice : double;
      pri profit : double; pri supp : Supplier; pri prod : Product;
      meth printProd ≙ ( • self.prod.printProd())
      meth printData ≙ (• print(self.acquisitionDate); print(self.purchasePrice);
            print(self.profit); print(self.supp.name); self.printProd())
end
```

Figure 16. Classes *Product* and *Item* after the applcation of Rule 3 ⟨*Move Method*⟩.

ertheless we have not implemented all refactorings proposed in [8], we dealt with a significant subset of them. Each refactoring is implemented as a rule in CafeOBJ, allowing the construction of a transformation environment.

The use of CafeOBJ was of great value to this work. The implementation of ROOL grammar and laws imposed no problems. Thus, this work could also be regarded as a contribution to the CafeOBJ community, as it could be considered as a programming transformation case study, in the context of this rewriting system. On the other hand, for the refactoring community, this work suggests that rewriting systems could be considered as supporting tools to construct refactoring

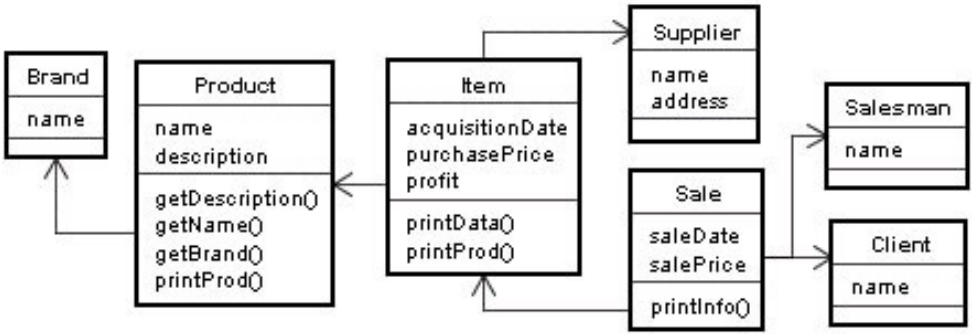


Figure 17. Class diagram after retactorings.

environments.

The majority of refactoring rules and laws of classes includes side conditions. When deriving the refactoring rules, we considered that these conditions are satisfied. Nevertheless, to construct a refactoring environment using a rewriting system such as CafeOBJ, we need to provide an efficient way to check the validity of these conditions in an arbitrary program, before applying the reduction. This is a challenge task, as many condition that appear in refactoring rules rely on information provided by the ROOL type system. In order to implement these conditions in CafeOBJ, we also need to describe the ROOL type system or, at least, to obtain type information from the program. This is our immediate future work.

The case study introduced in this paper is simple enough to validate the work. To develop some realistic and more complex case studies is also a direction of future work.

Acknowledgement

This work is partially supported by the Brazilian research agency CNPq.

References

[1] Arnold, K. and J. Gosling, "The Java Programming Language," Addison-Wesley, 1996.

[2] Back, R. J. R., Procedural Abstraction in the Refinement Calculus, Technical report, Department of Computer Science, Abo - Finland (1987).

[3] Cavalcanti, A. L. C. and D. Naumann, *A weakest precondition semantics for refinement of object-oriented programs*, IEEE Transactions on Software Engineering **26** (2000), pp. 713–728.

[4] Cavalcanti, A. L. C. and D. A. Naumann, *A weakest precondition semantics for an object-oriented language of refinement*, in: J. M. Wing, J. Woodcock and J. Davies, editors, *FM'99 - Formal Methods, Volume II*, number 1709 in Springer LNCS, 1999, pp. 1439–1459.

[5] Cavalcanti, A. L. C., A. Sampaio and J. C. P. Woodcock, *An Inconsistency in Procedures, Parameters and Substitution in the Refinement Calculus*, Science of Computer Programming **33** (1999), pp. 87–96.

[6] Cinnéide, M. Ó. and P. Nixon, *Automated application of design patterns to legacy code.*, in: *ECOOP Workshops*, 1999, p. 176.

[7] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, "Maude Manual (Version 2.1)," SRI International (2004).

[8] Cornélio, M., "Refactoring as Formal Refinements," Ph.D. thesis, Universidade Federal de Pernambuco (2004).

[9] Dijkstra, E. W., "A Discipline of Programming," Prentice Hall, 1976.

[10] Fowler, M., "Refactoring: Improving the Design of Existing Code," Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[11] Goguen, J., T. Winkler, J. Meseguer, K. Futatsugi and J. Jouannaud, *Introducing OBJ*, in: J. Goguen, editor, *Applications of Algebraic Specification using OBJ*, Cambridge, 1993 .

[12] Lano, K., J. Bicarregui and S. Goldsack, *Formalising design patterns*, in: D. Duke and A. Evans, editors, *1st BCS-FACS Northern Formal Methods Workshop, Ilkley, UK*, Electronic Workshops in Computing, 1996.
URL http://www.ewic.org.uk/ewic/workshop/view.cfm/NFM-96

[13] Larman, C., "Applying UML and Patterns," Prentice Hall, 2002.

[14] Lira, B. O., A. L. C. Cavalcanti and A. C. A. Sampaio, *Automation of a Normal Form Reduction Strategy for Object-oriented Programming*, in: *Proceedings of the 5th Brazilian Workshop on Formal Methods*, 2002, pp. 193–208.

[15] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, 1997, 2 edition.

[16] Morgan, C. C., "Programming from Specifications," Prentice Hall, 1994, 2 edition.

[17] Morgan, C. C., K. A. Robinson and P. H. B. Gardiner, On the Refinement Calculus, Technical Monograph PRG-70, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK (1988).

[18] Nakagawa, A., T. Sawada and K. Futatsugi, "CafeOBJ User's Manual - ver. 1.4.2, 1999. Disponível em http://www.ldl.jaist.ac.jp/cafeobj/doc/," (1999).
URL http://www.ldl.jaist.ac.jp/cafeobj/doc/

[19] Opdyke, W. F., "Refactoring Object-Oriented Frameworks," Ph.D. thesis, Urbana-Champaign, IL, USA (1992).
URL citeseer.ist.psu.edu/article/opdyke92refactoring.html

[20] Silva, A. L., M. M. Menezes and L. Silva, *Using CafeOBJ to implement a reduction strategy in the context of hardware/software partitioning.*, Electronic Notes in Theoretical Computer Science **95** (2004), pp. 63–82.

[21] Tokuda, L., "Evolving Object-Oriented Designs With Refactorings," Ph.D. thesis, University of Texas at Austin (1999).