УДК 519.681.2

СПОСОБ АВТОМАТИЗАЦИИ ПРОЦЕССА РЕФАКТОРИНГА

В. А. Кузин,
аспирант
В. В. Бураков,
канд. техн. наук, доцент
Санкт-Петербургский государственный университет аэрокосмического приборостроения

Представлен способ автоматизации рефакторинга кода программы на языке Java при помощи языка спецификаций Maude, позволяющий проводить автоматическое доказательство корректности преобразованного рефакторингом кода, основываясь на семантике Java; создавать пользовательские рефакторинги на базе уже имеющихся.

Ключевые слова *— рефакторинг, формализация ПО, корректность ПО, Java, семантика.*

Введение

Рефакторинг — это изменение во внутренней структуре программного обеспечения (ПО) с целью облегчить понимание работы ПО и упростить его модификацию, не затрагивая наблюдаемого поведения ПО [1]. Наиболее часто под рефакторингом понимают процесс преобразования кода.

Как следует из определения, ключевым моментом в применении рефакторинга является сохранение поведения ПО. Для проверки данного свойства необходимо формально описать и программу, подвергающуюся преобразованию, и сам рефакторинг.

На сегодняшний момент существует множество различных способов формального описания рефакторинга и проверки сохранения поведения ПО после применения рефакторинга. Например, в работе [2] доказательством корректности преобразований кода является истинность набора пред- и постусловий состояния программы; в работах с использованием аппарата теории графов — это сравнение графа, полученного после преобразования программы и набора некорректных графов; в работе [3] для доказательства используются строгие законы языка ROOL.

Можно выделить следующие общие ***недостатки*** имеющихся способов формального описания рефакторинга.

1. Неполная спецификация целевого языка программирования как основа формализации рефакторинга, отсюда возможна некорректность проводимых преобразований кода.

2. Проведение доказательства корректности преобразований программы вручную, что трудоемко.

3. Отсутствие механизма для построения пользовательских рефакторингов, что ограничивает уровень применимости подхода.

На основе этих недостатков нами были сформулированы следующие ***требования*** к формальному описанию рефакторинга:

1) базирование на спецификации целевого языка программирования;

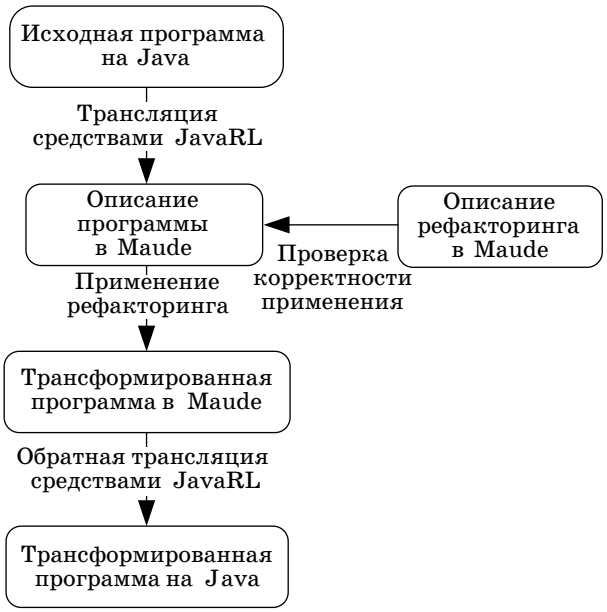
2) возможность автоматического доказательства корректности применения рефакторинга;

3) возможность создания пользовательских рефакторингов.

Язык спецификаций Maude

Исходя из представленных выше требований, нами предлагается способ описания рефакторинга, являющийся модификацией работы [4]. В рамках данного способа для формализации рефакторинга программ, написанных на объектно-ориентированном языке Java, используется язык спецификаций Maude — язык на основе логики переписывания (rewriting logic). В нем любое понятие, в том числе и рефакторинг, может быть описано тройкой (S, O, E), где S — множества (sorts) некоторых элементов; O — операторы (operators) над множествами; E — уравнения (equations), задающие производимые операторами преобразования.

В рамках проекта JavaFAN [5] для Maude в формальном виде была описана спецификация



■ ***Рис. 1.*** *Схема применения рефакторинга*

Java, что позволяет в полном объеме работать через Maude с семантикой Java (Maude Java Semantics — MJS): классами, переменными, объектами, их представлением в памяти и пр. Наше описание рефакторинга базируется на этой формализованной спецификации Java.

Maude является одновременно и интерпретатором, т. е. все Maude-описания по умолчанию являются исполняемыми. Поэтому описание рефакторинга в Maude можно использовать и для автоматизации преобразований целевой программы, и для автоматического доказательства корректности этих преобразований.

Рассмотрим подробнее, как выглядит ***применение рефакторинга*** к программе с учетом использования Maude.

На схеме (рис. 1) можно выделить следующие этапы.

1. Java-программа транслируется в Maude-описание при помощи средства JavaRL — транслятора кода Java в Maude-описание. Java-программа представляется в Maude в соответствии с описанием MJS. JavaRL и MJS являются экспериментальными разработками в рамках проекта JavaFAN [5].

2. К транслированной программе применяется рефакторинг, описанный на языке Maude. Каждый рефакторинг представляется отдельным модулем Maude. Пример описания рефакторинга приводится ниже.

3. Делаются проверки корректности применения рефакторинга к программе. Если проверки пройдены, программа трансформируется, иначе возвращается неизмененная программа. Этот пункт является ключевым: логика переписывания, на которой базируется Maude, и описание MJS, лежащее в основе каждого рефакторинга, обеспечивают формальную корректность проведенных над программой преобразований.

4. Программа транслируется обратно из кода Maude в код Java.

Концепция мини-операторов

Для предоставления возможности создавать пользовательские рефакторинги и облегчения работы по формальному описанию рефакторингов вводится концепция мини-операторов.

Мини-операторы описывают часто повторяемые в процессе рефакторинга действия над определенными элементами целевого языка программирования (классами, функциями и т. д.). Важным свойством мини-операторов является сохранение поведения программы, к которой они применяются. Это достигается за счет реализации каждого оператора в рамках описания MJS.

Мини-операторы делятся на три группы: операторы запроса, проверки условий и мини-трансформаций. Непосредственно при описании рефакторинга используются те или иные операторы из каждой группы.

Ниже приводятся сигнатуры некоторых операторов на языке Maude. Реализация каждого оператора, основанная на MJS, здесь не приводится.

Операторы запроса (рис. 2, *а*) получают определенный элемент целевого языка программирования (например, класс, метод или поле) по набору задаваемых параметров, например по имени элемента.

```
op getMethod : Qid1 Qid2 Types -> ClassMembers
```

Оператор получения метода с сигнатурой Qid2 Types из класса с именем Qid1. Возвращает список ClassMembers. Так сделано для случая, если будет возвращен identity-элемент noMember, т. е. если искомого метода не окажется в классе (identity-элемент должен быть заключен в список). Identity-элемент — это аналог NULL-значений в объектно-ориентированном программировании.

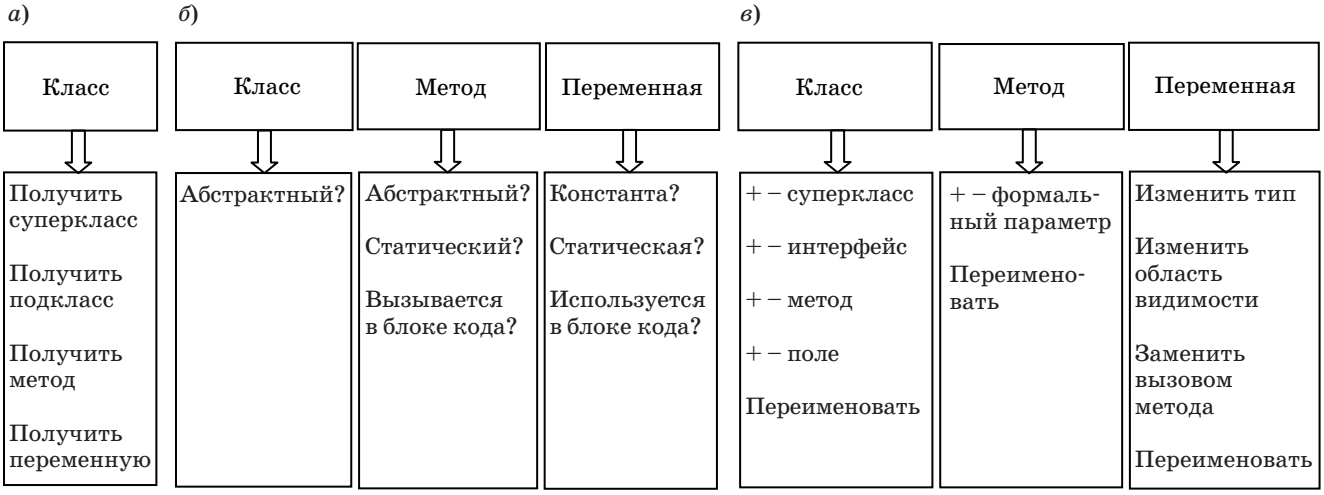
```
op getField : Qid1 Qid2 -> ClassMembers
```

Оператор получения поля с именем Qid2 из класса с именем Qid1. Возвращает список ClassMembers.

Операторы проверки условий (рис. 2, *б*) проверяют элемент целевого языка программирования на выполнение определенного условия.

```
op isStatic: ClassMember -> Bool
```

Проверяет, является ли член класса ClassMember статическим: анализирует, есть ли ключевое сло-



■ ***Рис. 2.*** *Общая схема операторов: а — запроса; б — проверки условий; в — мини-трансформаций («+» — добавить; «–» — удалить)*

во языка Java «static» в объявлении указанного поля или метода.

```
op isFinal: ClassMember -> Bool
```

Проверяет, является ли член класса ClassMember константным: анализирует, есть ли ключевое слово языка Java «final» в объявлении указанного поля или метода.

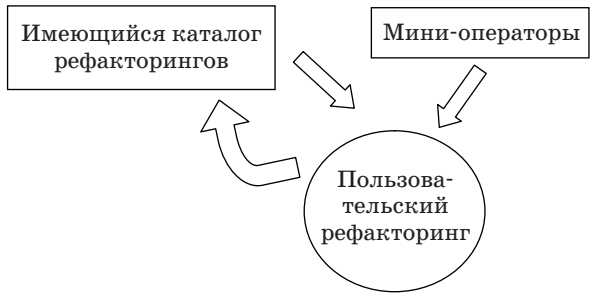
```
op usesField : Block Field -> Nat
```

Оператор проверки использования в блоке кода Block поля Field. Возвращает натуральное число, определяющее, как используется поле: 0 — не используется, 1 — пишется, 2 — читается.

Операторы мини-трансформаций (рис. 2, *в*) изменяют свойства определенного элемента целевого языка программирования.

```
op replaceFieldUsageWithMethod : Block Field Method -> Block
```

Оператор заменяет все обращения к полю Field в блоке кода Block на вызов метода Method. Возвращает трансформированный блок кода.



■ ***Рис. 3.*** *Общая схема создания пользовательского рефакторинга*

Пользовательские рефакторинги. Описанная выше концепция мини-операторов позволяет создавать пользовательские рефакторинги (рис. 3).

Пользователь может на основе уже описанного в Maude каталога рефакторингов и набора мини-операторов создавать новые рефакторинги. При этом новые рефакторинги добавляются в общий каталог рефакторингов.

Пример описания рефакторинга в Maude

Описание каждого рефакторинга является отдельным модулем Maude. В качестве примера ниже приводится описание рефакторинга «Инкапсулировать Поле» (Encapsulate Field) [1]. Данный рефакторинг заменяет все прямые обращения к указанному полю определенного класса на обращения через методы set и get (так называемые сеттеры и геттеры).

В листинге ниже жирным шрифтом выделены мини-операторы.

```
fmod ENCAPSULATE-FIELD is
pr JAVA-REF . pr CLASS-REF-HELPERS .
op EncapsulateField : Qid Qid ->
  JavaClassesRefactoring .
eq Cls <- EncapsulateField(CN, FN)
= if precondsEncapsulateFieldHold(Cls, CN, FN)
  then applyEncapsulateField (Cls, CN, FN)
  else Cls fi .
op precondsEncapsulateFieldHold :
  Classes Qid Qid -> Bool .
eq precondsEncapsulateFieldHold (noClass,CN,FN) =
  false. ***1
eq precondsEncapsulateFieldHold (((
    md Class CN sp cb) Cls), CN, FN) ***2
= getField(CN, FN) =/= noMember ***3 and
```

```
  getGetter(CN, FN) == noMember ***4 and
  getSetter(CN, FN) == noMember ***4 and
  isStatic(FN) == false and ***5
  isFinal(FN) == false . ***5
op applyEncapsulateField :
 Classes Qid Qid -> Classes .
eq applyEncapsulateField (((
   md Cl CN sp cb) Cls), CN, FN) =
 applyEncapsulateField ((
   addSetterAndGetter(Cl, FN) Cls), CN, FN) .
eq applyEncapsulateField ((Cl Cls), CN, FN) =
 applyEncapsulateField ((Cl Cls),
   getMethodList(Cl), CN, FN) .
op applyEncapsulateField (
 Classes, Methods, Qid, Qid) -> Classes.
eq applyEncapsulateField (Cls, ((
   md Mt mn mtp mb) Mts), CN, FN) =
 if (usesField(mb, getField(CN, FN)) > 0)
   if (usesField(mb, getField(CN, FN)) == 1)
   then refresh(Cls, replaceFieldUsageWithMethod(
       mb, getField(CN, FN), getSetter(CN, FN)))
   else refresh(Cls, replaceFieldUsageWithMethod(
       mb, getField(CN, FN), getGetter(CN, FN)))
   fi .
 else Cls fi .
op refresh(Classes, Block) -> Classes
endfm
```

Сам рефакторинг описывается оператором EncapsulateField(Qid Qid). Первый параметр — имя класса, поле которого инкапсулируется; второй параметр — имя инкапсулируемого поля. Затем идет равенство, в левой части которого стоит оператор Cls <- EncapsulateField(CN, FN), показывающий, что данный рефакторинг применятся к набору классов Cls состояния программы. В правой части равенства стоит условный оператор: если выполняется набор предусловий рефакторинга, описываемый оператором precondsEncapsulateFieldHold(Cls, CN, FN), то рефакторинг применяется к программе с помощью оператора applyEncapsulateField(Cls, CN, FN); если предусловия не выполняются, возвращается исходный набор классов Cls состояния программы. В приведенном листинге цифрами помечены предусловия в соответствии с номером условия в списке, приведенном ниже:

1. Набор классов конфигурации не пуст

```
precondsEncapsulateFieldHold (noClass,CN,FN) = false
```

Здесь noClass — identity-значение списка элементов Classes.

2. Существует класс с заданным именем

```
precondsEncapsulateFieldHold (((md Class CN sp cb) Cls), CN, FN)
```

3. В этом классе существует поле с заданным именем

```
getField(CN, FN) =/= noMember
```

Здесь noMember — identity-значение, возвращаемое мини-оператором getField(...), если поля с заданным именем нет в данном классе.

4. Не существует методов get и set для данного поля в данном классе

```
getGetter(CN,FN)==noMember and
getSetter(CN,FN)==noMember
```

5. Инкапсулируемое поле не является статическим или константным

```
isStatic(FN) == false and isFinal(FN) == false .
```

Доказательство корректности применения рефакторинга

Чтобы запустить MJS-представление Java-программы на выполнение в Maude, используется запись run(Cls E), где Cls — набор классов программы, E — некоторое выражение. run(Cls E) создает начальное состояние, включающее в себя Cls и континуацию, в которой E будет следующим к исполнению выражением. Континуацию можно понимать как стек, в который помещаются выполняемые инструкции. Результатом выполнения программы будет значение атрибута out финального состояния программы.

Ниже для наглядности доказательство приводится по шагам. В реальности Maude выполняет доказательство автоматически.

Теорема. Применение рефакторинга «Инкапсулировать Поле» не меняет поведения программы. Иначе говоря:

```
run(Cls E) = run((Cls <- EncapsulateField(CN, FN)) E)
```

где Cls : Classes — все классы программы; E : Exp — следующее к выполнению выражение в программе; CN, FN : Qid — имя класса и поля в этом классе, к которому применяется рефакторинг.

Доказательство. Если оператор precondsEncapsulateFieldHold возвращает false, то возвращается неизменный набор классов Cls. Следовательно, в данном случае теорема доказана. Если же precondsEncapsulateFieldHold возвращает true, то тогда известно, что в Cls есть класс с именем CN, содержащий нестатическое, неконстантное поле с именем FN и не имеющий для данного поля методов доступа (сеттера и геттера). Доказательство осуществляется последовательным применением равенств из листинга выше:

```
Cls <- EncapsulateField(CN, FN) = applyEncapsulateField (((md Cl CN sp cb)
Cls),CN,FN) = applyEncapsulateField(( addSetterAndGetter(Cl, FN) Cls), CN, FN) =
*
```

На данном шаге по имени CN находится класс Cl, поле FN которого инкапсулируется, и в этот

класс при помощи мини-оператора addSetterAndGetter(...) добавляются методы доступа к этому полю. Далее для каждого класса состояния программы, включая Cl, выполняется следующая последовательность:

```
* = applyEncapsulateField ((Cl Cls), getMethodList(Cl), CN, FN) =
applyEncapsulateField (Cls, ((md Mt mn mtp mb) Mts), CN, FN) = *
```

На ее выходе определено тело mb каждого метода Mt каждого класса Cl. Затем проверяется условие:

```
* = if (usesField(mb, getField(CN, FN)) > 0)
if (usesField(mb, getField(CN, FN)) == 1)
then refresh(Cls, replaceFieldUsageWithMethod(
  mb, getField(CN, FN), getSetter(CN, FN)))
else refresh(Cls,replaceFieldUsageWithMethod(
  mb, getField(CN, FN), getGetter(CN, FN))) fi.
   else Cls fi .
```

В случае, если в указанном теле идет обращение к полю FN класса CN, тело метода модифицируется: если значение этого поля изменяется, обращение к полю заменяется при помощи мини-оператора replaceFieldUsageWithMethod(...) на вызов сеттера; если значение поля читается — на вызов геттера. Оператор refresh обновляет список классов Cls состояния программы, обновляя измененное тело метода в нужном классе.

Как можно заметить, здесь все преобразования производятся только мини-операторами, а значит, поведение программы после применения данного рефакторинга остается неизменным.

Заключение

В работе представлена концепция по автоматизации рефакторинга кода на языке Java при помощи языка спецификаций Maude. Было введено и рассмотрено понятие мини-операторов, облегчающее формальное описание рефакторингов и позволяющее создавать пользовательские рефакторинги. В качестве примера дано описание рефакторинга Encapsulate Field на языке Maude. Через последовательное применение к программе равенств из данного описания рефакторинга показано, что рефакторинг проводится корректно.

В дальнейшем предполагается выделить на основе семантики языка Java полный набор мини-операторов по работе с кодом, описать их в Maude. Описать при помощи мини-операторов каталог рефакторингов Фаулера [1] в Maude. Исследовать возможность применения композиции рефакторингов к программе. Построить инструмент для ведения пользователем каталога рефакторингов, а именно создания своих собственных рефакторингов на основе набора мини-операторов и имеющегося базового каталога.

Литература

1. **Фаулер М.** Рефакторинг: улучшение существующего кода: Пер. с англ. СПб.: Символ-Плюс, 2003. 432 с.
2. **Garrido A., Meseguer J.** Formal Specification and Verification of Java Refactorings: Proc. of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation / IEEE Computer Society. 2006. Vol. 6. P. 165–174.
3. **Roberts D. B.** Practical Analysis For Refactoring: PhD Thesis. Urbana: University of Illinois at Urbana-Champaign, 1999. 127 p.
4. **Cornelio M. L.** Refactoring As Formal Refinements: PhD Thesis. Recife: Federal University of Pernambuco, 2004. 307 p.
5. **Farzan A., Chen F., Meseguer J., Rosu G.** Formal Analysis Of Java Programs In JavaFAN // Lecture Notes in Computer Science. Springer, 2004. Vol. 3114. P. 501–505.