Delft University of Technology
Software Engineering Research Group
Technical Report Series

Implementing Refactorings in the Spoofax Language Workbench

Maartje de Jonge and Eelco Visser

Report TUD-SERG-2013-008

TUD-SERG-2013-008

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Implementing Refactorings in the Spoofax Language Workbench

Maartje de Jonge and Eelco Visser

Dept. of Software Technology, Delft University of Technology, The Netherlands

Abstract. Spoofax is a language workbench for efficient development of textual domain-specific languages together with state-of-the-art IDE support. Spoofax integrates language and IDE development into a single environment, using concise, declarative specifications for languages and IDE services. We are extending Spoofax with a framework for the implementation of refactorings. The current paper gives an overview of the framework and demonstrates the implementation of refactorings for languages developed using Spoofax.

1 Introduction

Refactorings are behavior preserving structural transformations with the objective to improve the design of existing code [5]. Refactoring tools offer support for a set of predefined refactorings that are frequently applied by programmers, examples are: Rename, Extract method and Move method. Refactoring tools automate source code modifications and report errors and warnings for possible behavioral changes. The implementation of refactorings is challenging since different concerns must be handled, e.g., user interaction, source code modifications and behavior preservation checks.

IDE platforms such as Eclipse or Visual Studio reduce the effort to implement refactorings by factoring out common functionality into generic framework components. The Eclipse Language Toolkit [6] (LTK) provides a framework for the implementation of refactorings in Eclipse-based IDEs. The framework offers an API for implementing refactorings on top of a language neutral layer with components such as: wizards that guide the user through the refactoring process, change objects that represent the textual changes, and a viewer to visualize patches. The LTK also takes care of "undo" management.

Spoofax [10] is an Eclipse based environment for the development of languages together with full-featured IDE support. To support language development, Spoofax combines multiple domain-specific meta-languages, i.e., languages that target the domain of language engineering. The modular, declarative syntax definition formalism SDF [7,15] is used to specify the syntax of a language; the Stratego transformation language [16] provides a uniform formalism for concise specification of analysis, transformation, and code generation; editor descriptor DSLs [10] allow to configure editor services based on the syntax and semantics specified for a language; finally, Spoofax incorporates a language-parametric testing language (LPTL) [9] for the declarative specification of test cases for language definitions.

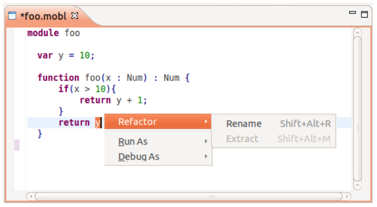
To reduce the effort to implement refactoring support for new languages, we are extending Spoofax with a refactoring framework build on top of the Eclipse LTK framework. The framework makes it possible to use the DSL-based Spoofax approach for the specification and implementation of refactorings. Furthermore, the framework provides language-parametric components to handle generic refactoring tasks. The framework incorporates the text reconstruction algorithm and the name binding preservation criterion discussed in respectively [1] and [2]. The current paper presents an overview of the framework and discusses how it can be used to implement refactorings for new languages. All examples in this paper use the Mobl language [8] as a target language.

Outline The paper is organized as follows. First, in Section 2 we discuss the workflow that is typically implemented by refactoring tools. Then, in Section 3 we show how the components from this workflow can be configured using a declarative specification language. Next, in Section 4 we give a short introduction into the Stratego transformation language, which is used in Section 5 to implement structural transformations on the AST, and in Section 6 to implement conditions for behavior preservation. Finally, Section 7 discusses automated testing of refactorings, using a language-parametric testing language (LPTL).

2 Tool Support for Refactorings

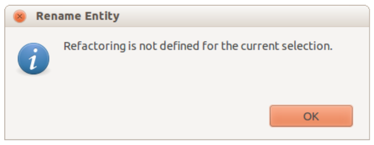
Refactoring tools offer support for a set of predefined refactorings, each of which implements a complex workflow that involves the user in making decisions and providing additional information. This section gives a short overview of the subsequent steps that are typical for refactoring workflows. All the discussed steps are supported in the Spoofax refactoring framework.

Refactor menu To apply a refactoring, the user first selects the code fragment where the refactoring takes place, and then chooses the appropriate refactoring from the refactor menu. In the given example, two refactorings are defined: Rename and Extract. The extract refactoring is disabled, since it is not defined on the current selection. As an alternative to the refactoring menu, refactorings can also be selected using a shortcut.



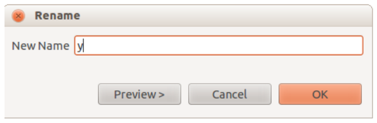
```
module foo

  var y = 10;

  function foo(x : Num) : Num {
      if(x > 10){
          return y + 1;
      }
      return y
  }
```

Initial failure dialog An error dialog box is shown in case the initial conditions of the refactoring are not met. This can happen in case the refactoring is not defined for the selected construct, or when the parser fails to construct an abstract syntax tree because of unresolved syntactic errors.



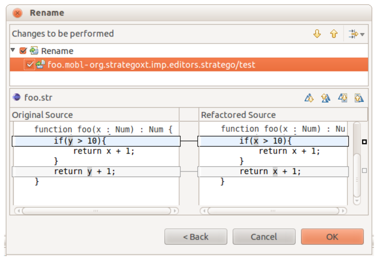
Refactoring is not defined for the current selection.

Input dialog After the initial validation is passed, a dialog is displayed that prompts the user to confirm the refactoring and supply additional information. The user can choose to apply the changes directly on the source code, or to first preview the changes in a preview dialog.



New Name: y

Preview dialog The preview dialog visualizes the textual changes that follow from the structural transformation. After inspecting the changes, the user can choose to apply the changes on the source code or to cancel the refactoring.



Changes to be performed
Rename
foo.mobl - org.strategoxt.imp.editors.stratego/test
foo.str

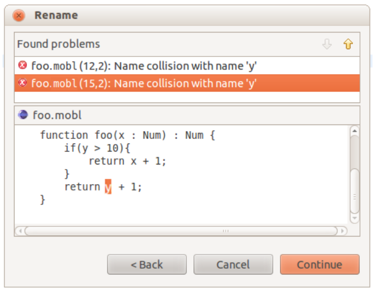
Original Source

```
function foo(x : Num) : Num {
    if(y > 10){
        return x + 1;
    }
    return y + 1;
}
```

Refactored Source

```
function foo(x : Num) : Nu
    if(x > 10){
        return x + 1;
    }
    return x + 1;
}
```

Error dialog Refactorings are supposed to preserve the behavior of the refactored program. However, in some cases the behavior preservation cannot be guaranteed. The error dialog presents errors and warnings for possible problems. After reviewing the provided information, the user decides whether or not to continue the refactoring.



Found problems
foo.mobl (12,2): Name collision with name 'y'
foo.mobl (15,2): Name collision with name 'y'
foo.mobl

```
function foo(x : Num) : Num {
    if(y > 10){
        return x + 1;
    }
    return y + 1;
}
```

At the end of the refactoring process, the structural changes are applied to the source code. The refactoring framework takes care of preserving the original layout and formatting newly inserted elements. Since the Spoofax refactoring framework is build on top of Eclipse LTK, the framework also offers Undo and Redo functionality, integrated with the Undo/Redo operations provided by the editor.

3 Declarative Specification of Refactorings

Spoofax comes with a family of editor service descriptor languages to define editor services. The description of an editor service configures its user interface

```
refactorings

  pretty-print: pp-mobl-string

  refactoring Id: "Rename" = rename-refactoring (cursor)
  shortcut: Shift + Alt + R
  input
    identifier : "New Name" = ""
```

Fig. 1. Declarative specification of a refactoring in the editor descriptor language.

aspects and specifies which transformation rule to apply for services that are implemented as AST transformations. The transformation rules themselves are specified in the Stratego transformation language [16]. In this section we introduce the descriptor language for the configuration of refactorings (Section 3.1), and we describe the signature of the transformation rule that implements the refactoring service (Section 3.2).

3.1 Refactoring Configuration Language

To illustrate the descriptor language for the refactoring service, Figure 1 shows an example specification. The `pretty-print` element is commonly defined for all refactorings and specifies the pretty-print strategy that is used to construct the text for newly inserted elements. The `refactoring` keyword indicates the specification of an individual refactoring. Below we give a brief explanation of the different elements that are part of this specification.

The `Id` element in Figure 1 indicates that the given refactoring is only specified on elements of the grammatical sort “Id”. Refactorings can be specified on multiple sorts separated by a comma, also, (non-empty) list sorts can be specified using the suffix “+”, as in `Stm+`. The grammatical sort can be refined with a constructor name, e.g., `Expr.FunCall`. The specified sorts determine whether the refactoring is enabled in the refactor menu, that is, the refactoring is enabled if and only if the selected construct is of the given grammatical sort.

The `"Rename"` element in Figure 1 sets the title of the refactoring. The title appears in the refactor menu and in the dialog boxes that handle the user interaction. As an alternative to the refactor menu, refactorings can also be called via a shortcut specified by the `shortcut` element.

The `input` element configures the user input dialog, which can contain different kind of input fields. A `text` input specifies a general text input field, while a `boolean` input specifies a checkbox that represents a boolean value. The `identifier` input shown in Figure 1 configures a text input whereby the input value must be a valid identifier; that is, it must match an identifier regular expression, while reserved keywords are excluded. The identifier pattern is looked up from the editor descriptor language definition, while the list of keywords is extracted from the grammar. Any problems are reported directly in the input

```
rename-refactoring:
  (user-input, selected, position, ast, path, project-path) →
  (ast-changes, fatal-errors, errors, warnings)
  where
    ...
```

Fig. 2. Refactoring transformation in Stratego.

dialogue box and disable the OK button. All input fields are specified according to the following schema:

```
<input-type> : "<label-text>" = "<default-value>"
```

Currently, default values can only be specified in the form of a literal string. As an improvement, we are planning to support Stratego transformations that calculate the value from the selected construct. In the rename example, the old name of the construct can then be used as a default for the new name. As a further improvement, we are also planning to extend the list of supported controls with more advanced controls, for example an input element to alter the name and order of function parameters.

Refactorings can have a number of annotations. The `(cursor)` annotation specifies that the construct at the cursor must be considered as the selected construct in case the user did not select a code fragment. The `(source)` annotation indicates that the refactoring applies to the AST that results after parsing, instead of the desugared and decorated AST that results after semantic analysis. Finally, the `(meta)` annotation specifies that the refactoring should only be available to language engineers, i.e., not when the plugin is deployed to end-users.

The `rename-refactoring` element specifies the transformation rule that implements the refactoring. The transformation rule itself is implemented in Stratego. Below, we explain the signature of refactoring implementation rules in more detail.

3.2 Refactoring Implementation Rules

Refactorings are implemented as regular Stratego rewrite rules with a fixed signature that forms the interface between the IDE and the transformation specification. The signature is illustrated in Figure 2. The terms at the left-hand side collect information from the IDE, while the terms at the right-hand side provide information to the IDE, required to perform code manipulations and to report possible problems to the user.

The input term of the rule collects information required to apply the refactoring transformation. That is, the values entered by the user in the input dialog (`user-input`), the selected construct (`selected`) and its position (`position`) in the AST (`ast`) of the file being edited, the project-relative file system path (`path`) and the file system path of the project itself (`project-path`).

The rule transforms this input into a new abstract term that provides the required information to calculate the textual changes, and to report semantic errors and warnings to the user. `ast-changes` provides a list of term changes that result after the refactoring. The term changes are specified as tuples consisting of the term before and after transformation and can be distributed across multiple files. The term changes are automatically translated into textual changes by the text reconstruction algorithm described in [1].

The terms `fatal-errors`, `errors` and `warnings` provide lists of found problems of decreasing severity. `fatal-errors` indicate that the selected transformation cannot be applied, thereby prohibiting the continuation of the refactoring. `errors` indicate known violations of semantic behavior. The user can review the information and choose either to abort the transformation, or to apply the transformation and manually fix the problems. `warnings` inform the user about possible behavioral changes or coding style violations. Again, the user decides whether or not to continue the refactoring process. The problems are specified as tuples consisting of the term to which the problem is attributed and an error message. The error dialog presents the error message and the error context to the user, using origin-tracking [3,10] to extract the location information from the error term.

4 The Stratego Transformation Language

Stratego [17] is a language for the specification of program transformations and analyses, based on the paradigm of term rewriting with programmable traversal strategies. This section gives a short introduction into the Stratego transformation language which is used in Spoofax to implement the transformations and analysis used in compilers and editor services.

Stratego uses conditional *rewrite rules* to define basic transformations on terms. These rules adhere to the following schema:

```
r : p1 → p2 where c
```

The rule `r` applies to a term when its left-hand side `p1` matches the term, and the (optional) condition `c` succeeds. The result is the instantiation of `p2` with the variable bindings found during pattern matching in `p1` and `c`. The rule is said to *fail* when either the subject term does not match the left-hand side or when the condition fails.

Rules are basic strategies that perform the transformation specified by the rule or fail. Strategies can be parameterized with strategy and term arguments, e.g., `r(s1 ... sm|t1 ... tn)`. Furthermore, strategies can be overloaded. That is, when invoking a rule with a given signature, all rules with that signature are tried in some unspecified order until one succeeds.

Strategies can be combined into more complex strategies by means of strategy operators. Sequential operators combine strategies that apply to the root of a term, examples are: identity (`id`), failure (`fail`), sequential composition (`s1 ; s2`), choice (`s1 + s2`), guarded choice (`s1 < s2 + s3`), negation (`not(s)`), and recursive closure (`rec x(s)`). Term traversal operators, e.g., `all(s)`, `one(s)`, and `some(s)`, express strategy application to the direct sub-terms of a term.

Combining these operators allows the generic definition of a wide range of term traversals. For example, `bottomup(s) = all(bottomup(s)); s` generi-

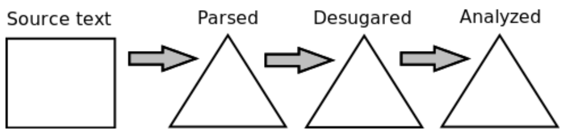


Fig. 3. Staged transformation: source code is parsed, desugared, and analysed. The resulting tree is used for semantic feedback, source-to-source transformations and code generation.

cally defines a post-order traversal. The Stratego standard library provides a collection of such strategies for general use.

5 Refactoring Transformations in Stratego

Spoofax employs a staged architecture for the implementation of compilers and language-specific editor services. The different stages are illustrated in Figure 3. First, a program is parsed to construct an abstract representation of the program. Then, the abstract representation is optionally simplified by desugaring, i.e., mapping "sugared" constructs in an enriched language to equivalent constructs in the core language. Finally, the desugared tree is semantically analyzed to detect name bindings and types. The result is an abstract syntax tree that is decorated with name binding annotations which ensure globally unique names. Other semantic information such as type information is stored in a global data structure which can be queried via the unique names. The implementation of semantic analysis falls outside the scope of this paper, a detailed description can be found in [10] and [11].

Refactorings require transformation and analyses on the abstract syntax tree. The specification of refactorings is considerably simplified by desugaring, since the transformation and the semantic analyses only needs to be implemented on the core syntax. Furthermore, to perform context-dependent transformation steps and to check semantic preservation conditions, refactoring transformations require access to semantic information such as types and name bindings. We therefore implement refactoring transformations on the AST that results after the semantic analysis stage.

In this section we discuss the implementation of two example refactorings, namely Rename and Extract method. We choose these refactorings since they are frequently applied by programmers and because they capture transformation patterns that are generically applicable to different languages. This section

```
module Example

  var x = 1;

  function incr(x : Num) : Num
  {
    return x + 1;
  }
```

```
Module(
  "Example"{"n9"}
, [ VarDecl("x"{"n10"}, Num("1"))
  , Function(
      []
    , "incr"{"n11"}
    , [FArg(
         "x"{"n12"},
         SimpleType("Num"{"n2"})
      )]
    , SimpleType("Num"{"n2"})
    , [Return(BinMethodCall(
         Var("x"{"n12"}),
         "+",
         Num("1")))])])
```

Fig. 4. Parsing, desugaring, and semantic analysis of a source fragment (left), results in a desugared and decorated AST (right).

focuses on the AST transformation, while the implementation of behavior preservation conditions is discussed in Section 6.

5.1 Rename

The Rename refactoring is essentially a “smart” search and replace. It allows users to change names of program entities such as classes, methods, fields, and variables. The renaming can be called at declaration sites and at call sites, furthermore, all corresponding references in the code must be updated automatically. Since different program entities may accidentally have the same name, the name of an identifier is not sufficient to determine its reference. The implementation challenge for the rename transformation is to determine which identifiers must be renamed given a certain selected name.

Determining the reference of identifiers requires information about the binding structure of the program at hand. In Spoofax, the binding structure of the abstract syntax tree after analysis is made explicit by using name annotations that are globally unique; that is, two identifiers are annotated with the same reference name if and only if they bind to the same declaration. For example in Figure 4, the field declaration `x` in `var x = 1;` is distinguished from the function parameter `x` in `x : Num` by the annotations, `"x"{"n10"}` respectively `"x"{"n12"}`. The variable access `x` in `return x + 1;` refers to the function parameter `x : Num`, which is made explicit by the annotation `"x"{"n12"}`.

Globally unique name annotations make the implementation of the rename refactoring almost trivial, Figure 5 shows the Stratego code. The `alltd` strategy applies the (anonymous) rewrite rule `oldname -> newname` along a frontier of the `ast` term, replacing all terms that match the old name (including its annotation) with the new name. The name annotation is preserved in the new name term; this allows checking of name binding preservation as discussed in Section 6.1. Thus, application of the `rename` rule with the parameters `"x"{"n12"}`

```
rename(|oldname, newname):
  asts → <alltd(preserve-annos((oldname → newname)))> asts
```

Fig. 5. Rename refactoring transforms `oldname{ref}` terms to `newname{ref}` terms. The `{ref}` annotation distinguishes between different identifiers that accidentally have the same name.

and `"y"` on the AST of Figure 4 results in an AST whereby both `"x"{"n12"}` terms are replaced with `"y"{"n12"}` terms. Notice that the transformation rule of Figure 5 does not contain any language-specific elements, which means that it is generically applicable to different languages.

Global variable declarations may be referenced across multiple files. Furthermore, the renaming of a global variable may accidentally affect the name bindings of other global variables that have the same name as the newly inserted name. For performance reasons, it is important to restrict the set of analyzed and transformed files to a set of files that are possibly affected by the rename transformation. In Spoofax, name binding information is stored in an external data structure that can be queried for files that contain the definition or use sites of a given reference name. We use this infrastructure to collect the ASTs of files that are affected and/or possibly semantically endangered by the transformation. The name analysis and resulting data structure are described in [11].

5.2 Extract Method

We describe the Extract method refactoring in three parts. First, we focus on the basic transformation, ignoring parameters and return values. Then, we show how parameters and return values can be calculated based on a data-flow analysis. Finally, we discuss which parts of the extract method refactoring can be implemented generically for different languages.

Basic Transformation Extract method is a refactoring that encapsulates a previously anonymous list of statements in a newly created method. The extracted method presumably represents a well-defined piece of functionality which can potentially be reused by other methods. The basic transformation for method extraction involves the following two steps. First, the selected statements are separated out into a new method which is inserted after the method where the abstraction takes place. Secondly, the selected statements are replaced by a call to the extracted method. The Stratego code for the basic transformation is shown in Figure 6.

Data-Flow Repair Strategies The basic transformation of Figure 6 is incomplete since it does not compensate for possible changes in the use-define chains of local variables. We identified two problems for which we developed two different compensation strategies.

```
extract-block(|name, from, to):
  ast → ast-transformed
  where
    pos-method := <fetch-up-position(?Function(_,_,_,_)|ast)> from;
    pos-extracted := <position-next-sibling> pos-method;
    method-call := ExpStat(Call(name, []));
    selected := <select-sublist(|from, to)> ast;
    extracted-method := Function(name, [], None(), selected);
    ast-transformed := <
      replace-sublist(|selected, [method-call]);
      insert-list-element(|pos-extracted, extracted-method)
    > ast
```

Fig. 6. Basic transformation that encapsulates a statement block in a newly created method.

```
function funX() {  function funX() {   function funX() {
  var j : Num = 3;   var j : Num = 3;    var j : Num = 3;
  print(j);          print(j);           print(j);
  j = 5;             funY();             funY();
  foo(j);          }                   }
}
                   function funY() {   function funY() {
                     j = 5;              var j : Num;
                     foo(j);             j = 5;
                   }                     foo(j);
                                       }
```

Fig. 7. Basic extraction of `funY` causes a missing variable declaration in the extracted method (mid). The problem is fixed by inserting the missing declaration (right).

Missing declarations The extracted method or the remainder of the original method may contain variables that miss their original declaration. This problem occurs in case the selected statements contain accesses of variables that are declared outside the selection, or vice versa. Missing declarations that occur before a variable assignment can be compensated by inserting the declaration before the assignment, Figure 7 provides an example. The repair strategy for missing declarations that occur before a variable read access is given in the next paragraph, which discusses function parameters and return values.

We detect missing declarations by comparing declarations and uses before and after, respectively, the `remove-sublist` transformation for the remainder of the original method, and the `select-sublist` transformation for the body of the extracted method. In the latter case, we filter out declarations that are passed as a function parameter as described in the next paragraph. To obtain the declarations that are missing after a transformation, we first collect all variable declarations in the original term, and then exclude from this collection all declarations that remain after the transformation and all declarations that are not accessed in the transformed term. The implementation in Stratego is straight forward, taking as input a rule that maps declarations onto the name of the de-

```
function funX() {
  var j : Num = 3;
  print(j);
  foo(j);
  j = 5;
  bar(j);
}
```

```
function funX() {
  var j : Num = 3;
  print(j);
  funY()
  bar(j);
}

function funY() {
  foo(j);
  j = 5;
}
```

```
function funX() {
  var j : Num = 3;
  print(j);
  j = funY(j)
  bar(j);
}

function funY(j : Num) {
  foo(j);
  j = 5;
  return j;
}
```

Fig. 8. Basic extraction of `funY` affects the reaching definitions of the variable `j` in `bar(j)` and `foo(j)` (mid). Method parameters and return values compensate for the broken data flow (right).

clared variable. Since we take as input the analyzed abstract syntax, we assume that all variables are annotated with a unique name.

Parameters and return values The reaching definitions of a variable are given by all declarations and assignments that can reach the variable without an intervening assignment. The reaching definitions determine the values that the variable can have at a certain point in the program. The basic extract transformation may affect the reaching definitions of a variable in case a read access is moved to the extracted method while a preceding assignment or declaration remains in the original method, or vice versa. The remedy is to pass the variable value as a function argument or as a return value, respectively. Figure 8 provides an example. Some languages allow multiple return values, for example by packing the values into a tuple or in the form of output parameters. For other languages, only one value can be returned, which means that multiple return values must be reported to the user as a possible behavioral change.

We detect missing reaching definitions by comparing use-definition chains of local variables before and after the basic extract transformation. For this, we assume a (partial) data-flow analysis that annotates all local variable declarations and assignments with a unique identifier, and all variable read accesses with a list of identifiers that correspond to their reaching definitions. The data-flow analysis can be implemented on top of the name analysis and a possible control-flow analysis. The implementation of flow analyses in Stratego is discussed in [13].

We use data-flow annotations to detect missing reaching definitions, Figure 9 illustrates the process. First the original method is decorated with data-flow annotations (left fragment). Then, the original method is transformed by extracting the selected statements. The original data-flow annotations are preserved during the transformation and express the *intended* use-definition chains in the transformed constructs (mid fragment). Next, we re-analyze the data-flow of the transformed constructs which sets annotations that express the *actual* use-definition chains (right fragment). Finally, we compare the intended use-definition chains with the actual use definition chains. In the given example, this

```
function funX() {      function funX() {      function funX() {
  var j{d1} : Num = 3;   var j{d1} : Num = 3;   var j{e1} : Num = 3;
  print(j{[d1]});        print(j{[d1]});        print(j{[e1]});
  foo(j{[d1]});          funY()                 funY()
  j{d2} = 5;             bar(j{[d2]});          bar(j{[e1]});
  bar(j{[d2]});        }                      }
}
                       function funY() {      function funY() {
                         foo(j{[d1]});          foo(j{[]});
                         j{d2} = 5;             j{e2} = 5;
                       }                      }
```

Fig. 9. Comparison of the intended data-flow pattern (mid) and the actual data-flow pattern (right) shows a violation for the variable `j` in the original method (`d1` and `d2` both map to `e1`), and in the extracted method (`d1` is not matched).

reveals a violation in the remainder of the original method (`d1` and `d2` both map to `e1`) as well as in the extracted method (`d1` is not matched).

Figure 10 shows the Stratego code for the described process. The rewrite rule `extract-method-parameters` calculates the method parameters and return values that compensate for missing reaching definitions in respectively the extracted method body (`select-sublist`) and the remainder of the original method (`remove-sublist`). The rule `get-df-changes` compares the annotations that express the intended data-flow (`analyze-df; transform`), with annotations that express the actual data-flow (`transform; analyze-df`) on the term that results after the transformation. To detect the binding violations (`binding-violations`), we first construct a mapping between old and new identifiers set as single annotations at the declaration and assignment sites. We then try to apply this mapping to the annotation lists at the read accesses. A violation is detected in case the mapping cannot be applied. The code is left out of the figure. The data-flow preservation technique is similar to the name binding technique discussed in [2]

Reusable Extraction Strategies The implementation of the Extract method refactoring contains language generic elements which we factored out as part of the refactoring framework. First, the calculation of missing declarations and the calculation of method parameters and return values are language generic, taking the data-flow analysis as a strategy parameter. Furthermore, we generalized the basic transformation, taking the extracted method and method call (plus added declarations) as term parameters. The construction of the extracted method and the method call is left to the language developer.

6 Behavior Preservation Conditions

Refactorings are structural transformations that preserve the behavior of a program. Though it is hard to guarantee behavior preservation in general, refactoring tools can detect violations of the static semantics by means of a static analysis

```
extract-method-parameters(analyze-df):
  (method, from, to) → (parameter-vars, return-vars)
  where
    parameter-vars :=
      <get-df-changes(select-sublist(|from, to), analyze-df)> method;
    return-vars :=
      <get-df-changes(remove-sublist(|from, to), analyze-df) > method

get-df-changes(transform, analyze-df):
  term-before → df-changes
  where
    df-intended := <analyze-df; transform> term-before;
    df-actual   := <transform; analyze-df> term-before;
    df-changes  := <binding-violations> (df-intended, df-actual)

//Returns terms for which the binding annotation pattern has changed
binding-violations:
  (df-intended, df-actual) → binding-violations
  where
    ...
```

Fig. 10. Method parameters and return values are calculated by comparing annotations that express the intended data-flow (`analyze-df; transform`), with annotations that express the actual data-flow (`transform; analyze-df`) on the term that results after the transformation.

of the source code. Ad hoc precondition based approaches are tedious and error-prone, since it is hard to guarantee that the conditions implemented for behavior preservation implement the same static semantics as the compiler for the language. Instead, we aim for a more generic approach that reuses the language semantics to check preservation criteria. We propose a language-parametric criterion for the preservation of name bindings (Section 6.1) and for the preservation of static semantic correctness (Section 6.2). In Section 6.3 we discuss the preservation of data- and control-flow.

6.1 Name Binding Preservation

Name bindings associate identifiers with program entities such as variables, fields and methods. All refactorings that introduce new names into a scope have to guard against accidental changes of existing name bindings, which change the semantic behavior of the program. Figure 11 shows an example where a Rename refactoring transformation accidentally causes a name collision. As a result, the `foo` function before and after the refactoring returns a different value.

Name bindings form a semantic concern that should be preserved by refactorings. Intuitively, all name accesses in a program should bind to the same declarations before and after the transformation. Name binding preservation is checked using the language-parametric preservation criterion described in [2]. The preservation criterion takes as input the name analysis defined for the language and returns a (possibly empty) set of name binding violation errors. The violation errors are constructed as a tuple consisting of the offending identifier and a generic “Name collision detected for . . . ” message.

```
module Example          module Example

  var y = 0;              var x = 0;

  function foo() : Num    function foo() : Num
  {                       {
    var x = 1;              var x = 1;
    return y;               return x;
  }                       }
```

Fig. 11. Renaming `y` to `x` causes a name collision that changes the behavior of the program; the `foo` function in the left fragment returns 0, while the `foo` function in the right fragment returns 1.

```
application example       application example

  screen root() {           screen foo() {
    header("Example")         header("Example")
  }                         }
```

Fig. 12. Renaming `root` to `foo` causes a semantic error, since the root screen is a required element for applications written in Mobl.

6.2 Semantic Consistency Preservation

Semantic constraints are part of the specification of a language and define whether a program is semantically well-formed. Common examples include constraint errors for unbound variables, duplicate declarations, missing elements, and incompatible types. In addition to semantic errors, semantic warnings can be used to warn the user against code fragments that are probably not intended (e.g. dead code), or code fragments that are likely to result in runtime errors (e.g. uninitialized variables), or style violations (e.g. ignoring a style convention to use a capital).

Refactorings are supposed to preserve the semantic well-formedness of a program. Refactoring implementations must check this preservation criterion since the applied transformations may introduce semantic errors. Figure 12 provides an example; renaming the `root` screen in a Mobl [8] program results in an incorrect program. The resulting program violates the semantic constraint that the root screen is a required element.

Semantic errors and warnings are reported in an IDE by placing error and warning markers near the construct that caused the problem. In Spoofax, semantic constraints are defined as rewrite rules that succeed for terms that violate the constraint. The constraints are checked in a generic tree traversal, `collect-all(constraint-error, conc)`, that collects constraint errors for all terms where a `constraint-error` rule succeeds. The `constraint-error` rule produces a target term plus a diagnostic message that is reported to the user. The target term represents the term to which the error is attributed; from this

```
//Input term consists of the analyzed AST before the transformation
//and the (re)analyzed AST after the transformation.
consistency-problems(constraint-issue):
  (ast-before, ast-after) → (new-issues, solved-issues)
  where
    issues-before := <collect-all(constraint-issue, conc)> ast-before;
    issues-after  := <collect-all(constraint-issue, conc)> ast-after;
    new-issues    := <diff(same-issue)> (issues-after, issues-before);
    solved-issues := <diff(same-issue)> (issues-before, issues-after)

same-issue:
  issue-tuple@((trm1, message1), (trm2, message2)) → issue-tuple
  where
    <origin-equal> (trm1, trm2);
    <equal> (message1, message2)
```

Fig. 13. Language-parametric rule that returns all semantic issues that are introduced or solved by the refactoring.

term, the required location information is retrieved by the Spoofax infrastructure.

We reuse the constraint error rules defined in the compiler front end to implement a language-parametric preservation criterion for semantic consistency. First, we stretch the definition of preservation a bit so that it also applies to refactorings that take a semantically incorrect program as input. That is, we report consistency problems for all semantic errors that are introduced (or solved) by the refactoring. Figure 13 shows the Stratego code. The `consistency-problems` rule is used to detect semantic errors (or warnings) that are introduced or solved after the transformation. The rule returns all constraint issues that occur in the analyzed AST before the transformation (`ast-before`), but not in the analyzed AST after the transformation (`ast-after`), and the other way around. To see if two issues actually represent the same problem, the `same-issue` rule compares their error messages and the origin of their target terms, i.e., the terms in the original AST that originated the target terms in the AST before and after the transformation.

We combined the criteria for name binding preservation and semantic correctness preservation in a single rule that carefully schedules analysis and constraint checking so that they are applied in the right order and so that they are never applied twice on the same AST.

6.3 Data- and Control-Flow Preservation

Control-flow refers to the order in which the individual instructions or statements of a computer program are executed or evaluated. The control-flow determines the data-flow, i.e., the values that variables can have at various locations in the program. Refactorings that affect the structure of the abstract syntax tree risk changing the control- and/or data-flow of the program, which may result in a change of the observable behavior. Figure 14 provides an example whereby a

```
class A {                            class A {
  void foo() {                         void foo() {
    int i = 5;                           int i = 5;
    if(i > 0)                            bar(i);
      return;                            System.out.println(i);
    System.out.println(i); /+     +/   }
  }                                    void bar(int i) {
}                                        if(i > 0)
                                           return;
                                       }
                                     }
```

Fig. 14. Extracting a fragment with a return statement causes a control flow violation that changes the behavior of the program; the `foo` function in the left fragment would return immediately, while the `foo` function in the right fragment prints 5 before returning.

naive implementation of Extract method results in a behavioral change due to its effect on the control-flow.

The cause of the incorrect refactoring shown in Figure 14 is the `return` statement which cannot be moved safely into the newly constructed method. This symptom is typical for control statements that have a jump like nature; other examples include Java constructs such as `break`, `continue` and `try ...catch`.

As an ad hoc solution to guard against flow violations, refactoring implementations can create warnings for any transformation that restructures code fragments containing branching statements that endanger control-flow preservation. The user of the refactoring can review the information and decide whether or not to perform the refactoring. More refined conditions may be implemented to prevent warnings for situations that are in fact harmless, and to report errors instead of warnings for sure behavioral changes.

A more sophisticated approach is proposed in [4]. Given an existing control-flow and data-flow analysis, the authors propose an invariant-based preservation condition. The condition for control-flow preservation basically states that all statements in the affected methods should maintain their control-flow predecessors and successors throughout the refactoring. The data-flow condition states that all variables should have the same reaching definitions before and after the refactoring.

We implemented a generic data-flow preservation condition based on term annotations set by a (partial) data-flow analysis. In Section 5.2 we demonstrated how this preservation condition can be used in Extract method to calculate the required method parameters and return values of the extracted method. We did not (yet) look into the preservation of control-flow. Firstly, because most language implementations in Spoofax do not include flow analyses. Furthermore, since flow preservation is mostly a local problem we consider it acceptable to let the user decide, after reviewing the information provided by a warning. Still, it seems possible to extend the refactoring framework with a criterion for control-flow preservation based on control flow annotations.

7 Testing Refactorings

Testing is one of the most important techniques to control the quality of a piece of software, and to gain confidence in it working correctly. Programmers rely on refactoring tools to restructure their source code as expected, and to warn them against possible semantic changes. To meet the expectations of the users of refactoring tools, refactoring implementations must be tested.

7.1 Test Specification Language

Spoofax offers support for testing of language definitions and IDE services by means of a language-parametric testing language (LPTL) [9]. The testing language allows declarative specification of test cases, using language embedding to quote program fragments in the language under test. All test cases are specified according to the following schema:

```
test description [[
  fragment
]] condition*
```

where *description* is a string that describes the test case, *fragment* is an embedded program or program fragment in the subject language, and the *condition** elements specify expectations with regard to the outcome of actions performed to the input fragment. Conditions for test fragments are specified using an extensible set of test condition specification constructs. Currently, this set contains constructs for testing of syntax, static semantics, dynamic semantics, generated code and editor services. In the input fragment, a subfragment can be marked as "selected" by surrounding it with a pair of square brackets (`[[...]]`). This feature is useful to test editor services that depend on the user selection such as content completion, reference resolving and refactorings.

The testing framework offers tool support for editing and running the test cases. The tool support for editing tests includes editor services for the test specification language as well as for the language under test. The tool support for running tests includes live evaluation of test cases as they are edited and a batch test runner, which is particularly useful for running larger test suites.

7.2 Specifying Refactoring Tests

The language-parametric testing framework can be used to test refactoring implementations in Spoofax. Figure 15 provides an example of a refactoring test case. The example tests the rename refactoring with the input term "y" which represents the user input value, determining the new name of the selected identifier "x". The first test condition compares the output of the AST transformation to the expected AST. Both the input as the expected output program are specified as concrete syntax fragments, though the actual transformation and comparison is done on their abstract representations. The second test condition lists the expected problems. The semantic problems are specified as the number of

```
language mobl

test Rename Global-Variabele-Shadowing [[
  module Example

    var y = 1;

    function foo(x : Num) : Num {
      return [[x]] + y;
    }
]] refactor rename-refactoring("y") to [[
  module Example

    var y = 1;

    function foo(y : Num) : Num {
      return y + y;
    }
]]
1 error /Name collision at 'y'/
```

Fig. 15. Testcase for Rename refactoring in Mobl.

expected problems, followed by their type (`fatal-errors`, `errors`, `warnings`), plus (optional) a list of substrings that are part of the subsequent problem messages. The substrings are of the form `/.../`. The condition 0 `errors` can be omitted if no errors are expected, likewise for warnings and fatal-errors.

8 Conclusion

Spoofax [10] is an Eclipse based environment for the development of textual languages together with full-featured IDE support. We have extended Spoofax with a framework for the implementation of automated refactorings. The two pillars of the framework are the use of domain-specific meta languages to define the language-specific aspects of refactorings, and the use of language-parametric techniques to handle generic refactoring concerns. Together, these techniques help language engineers to implement refactoring tools that are functional and reliable for end users.

The presented refactoring framework focuses on the implementation of predefined refactorings for end users. An interesting direction for future work is support for the implementation of refactorings for and by language developers. Given the fact that language development in Spoofax involves multiple DSLs, this requires an integrated approach to cross-language analysis and refactoring [14]. Furthermore, the IDE support for refactorings must implement an open structure that allows the application of user-defined transformations [12].

References

1. M. de Jonge and E. Visser. An algorithm for layout preservation in refactoring transformations. In U. Aßmann and T. Sloane, editors, *Software Language Engi-*

neering, Fourth International Conference, SLE 2011, Braga, Portugal, July, 2011, Revised Selected Papers. Springer, 2012.

2. M. de Jonge and E. Visser. A language generic solution for name binding preservation in refactorings. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 2:1–2:8, New York, NY, USA, 2012. ACM.
3. A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545, 1993.
4. T. Ekman, M. Schäfer, and M. Verbaere. Refactoring is not (yet) about transformation. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 5:1–5:4, New York, 2008. ACM.
5. M. Fowler. Refactoring: Improving the design of existing code. volume 2418 of *Lecture Notes in Computer Science*, page 256. Springer, 2002.
6. L. Frenzel. The language toolkit: An api for automated refactorings in eclipse-based ides. *Eclipse Magazine*, 5, 2006.
7. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
8. Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In K. S. Fisher, editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, SIGPLAN Notices, Portland, Oregon, USA, 2011. ACM.
9. L. C. L. Kats, R. Vermaas, and E. Visser. Integrated language definition testing: Enabling test-driven language development. In K. S. Fisher, editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, SIGPLAN Notices. ACM, 2011.
10. L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 444–463. ACM, 2010.
11. G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Language-parametric name resolution based on declarative name binding and scope rules. In K. Czarnecki and G. Hedin, editors, *Software Language Engineering, Fourth International Conference, SLE 2012, Dresden, Germany, September, 2012, Revised Selected Papers*, 2013. (To appear).
12. H. Li and S. Thompson. Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 32–39, New York, NY, USA, 2012. ACM.
13. K. Olmos and E. Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In R. Bodk, editor, *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3443 of *Lecture Notes in Computer Science*, pages 204–220. Springer, 2005.
14. D. Strein, H. Kratz, and W. Lowe. Cross-language program analysis and refactoring. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 207–216, Washington, DC, USA, 2006. IEEE Computer Society.
15. E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.

16. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, 2004.
17. E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In M. Felleisen, P. Hudak, and C. Queinnec, editors, *Functional programming*, pages 13–26. ACM, 1998.

TUD-SERG-2013-008
ISSN 1872-5392