A Refactoring Constraint Language and its Application to Eiffel

Friedrich Steimann, Christian Kollee, and Jens von Pilgrim

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org, Christian.Kollee@feu.de, Jens.vonPilgrim@feu.de

Abstract. We generalize previous work on constraint-based refactoring and develop it into the definition of a constraint language allowing the specification of refactorings in a completely declarative way. We present a compiler that transforms specifications in our language to plug-ins for an IDE that, together with an accompanying framework providing the necessary infrastructure, implement the specified refactoring tools. We evaluate our approach by presenting specifications of three different refactorings for the Eiffel programming language, and by applying the resulting refactoring tools to several sample programs. Outcome suggests that our approach is indeed viable.

1 Introduction

Constraint-based refactoring has proven to be a powerful approach to mastering the complexity of various type and accessibility related refactorings [4, 9, 14, 15, 26, 29, 30]. The declarative nature of constraints and of the rules that generate them allow one to stay close to the language specification, and away from the combinatorial explosion of possible expressions whose correct handling makes imperative formulations of the preconditions and the mechanics of a refactoring such a daunting task. However, we observe that the scope of constraint-based refactoring has so far unnecessarily been constrained to treating a single aspect of a programming language — such as typing or accessibility — in isolation, where the same approach, with some extensions, could be used to solve a much broader class of refactoring problems, including the joint handling of different language aspects in one refactoring. Preparing the ground for such a broadening of scope is the aim of our work presented here.

More concretely, we count six main contributions in this paper (and ask the reader to accept the following enumeration as an outline also):

1. We generalize previous work on constraint-based refactoring, extending it to the control of properties beyond types and accessibilities, allowing their joint treatment within a single refactoring (Section 3).
2. We identify a number of challenges of constraint-based refactoring uniformly posed by different programming languages, providing solutions that remain entirely in the realm of constraint satisfaction and thus a single solution framework (Section 4).

3. We specify a powerful, yet concise constraint language, called REFACOLA, allowing the completely declarative specification of various refactorings (Section 5.1).
4. We present an implementation of that language allowing the generation of refactoring tools for various platforms directly from their specifications (Section 5.2).
5. We present the REFACOLA specifications of three declaration-related refactorings for the Eiffel programming language (Section 6), showing that the constraint-based approach is capable of dealing with problems rather different from those found in Java (to which constraint-based refactoring has so far exclusively been applied).
6. We evaluate the computational cost of constraint-based refactorings resulting from REFACOLA specifications by applying them to a body of Eiffel programs (Section 7), showing the feasibility of our approach.

We have chosen Eiffel for our evaluation, and thus decided to break with the Java monotony that currently characterizes the field, because (a) continued concentration on a single language tends to draw attention to accidental problems caused by that language's design, rather than the essential problems of refactoring, (b) Eiffel is sufficiently different from Java and its kin to support our claim of generalizing constraint-based refactoring, and (c) because the Eiffel standard [6] makes heavy use of so-called validity rules, a form of language specification that has a rather direct mapping to the constraint rules of constraint-based refactoring, thereby increasing one's belief in that refactoring tools correctly incorporate the language specification.

2 Motivation and Related Work

Although the discipline of refactoring — as initiated by the works of Griswold [11] and Opdyke [21] — is already almost two decades old, currently available refactoring tools are still plagued with bugs (see, e.g., [3, 22, 24–26]). While some would maintain that refactorings are used mainly in agile settings characterized by excessive regression testing anyway [8], we object that refactoring tools are *metaprograms* that play in the same league as editors, compilers, and version control, for the correctness of which one too would not want to rely on testing one's own *object programs*.

Roughly, the symptoms of today's refactoring tools' failures can be divided into four categories:

1. A refactoring that is possible is nevertheless refused.
2. The refactored program contains errors (compile time or runtime) the original program did not exhibit.
3. The refactored program behaves differently from the original program.
4. The refactored program does not exhibit the change of structure reflecting the refactoring intent.

Except for the last, all problems can be blamed on a partial ignorance, on behalf of the refactoring tool, of the target language's syntax and semantics, and thus should be resolvable, once and for all, by incorporating the relevant parts of the language specification into the tool's refactoring procedure.

One approach to making a refactoring language-specification aware is to specify it in terms of rewrite rules to be applied to a graph representation of programs [18],

where the rewrite rules are designed to observe the syntactic and semantic rules of the programming language. However, it turns out that the necessary precondition checking, and also the correctness-preserving transformations themselves, are nontrivial to formulate using graph grammars, which is why these approaches resort to considerable imperative components ("programmed graph grammars") that suffer the same problems as the standard procedural implementations of refactorings. As far as we know, no widely disseminated refactoring tools as yet utilize this approach.

ASTGen [3] uses comprehensive syntactic and (static) semantic knowledge of the programming language whose programs are to be refactored, for generating programs that provoke errors in refactoring tools. While this proved to allow for very effective testing of existing refactoring implementations, it is not obvious how the language-awareness of ASTGen could be incorporated directly into a refactoring tool.

In a series of works, Tip et al. [9, 15, 30, 29] have used constraints to include Java's typing rules into the precondition checking and mechanics of refactoring tools, thereby avoiding errors of above category 2. Ref. 26 has picked up the approach and transferred it to the rules of access control in Java. However, as has been pointed out in [26, 29], obeying the typing and accessibility rules alone is insufficient to prevent errors of category 3: in presence of overloading, hiding, and other name resolution issues, constraints modelling the name lookup in Java may be needed which, given its imperative nature, are hard to express. That name binding rules even for a language such as Java can be modelled as constraints (although admittedly at some expense) has been shown in [26, 27].

In a complementary series of works, Schäfer et al. [22–25] have elaborated how to preserve, among other dependencies, the binding of references (names) to declared entities. For this, they have established a notion of *locked names* which can be conceived of as pointers to declared entities persisting all refactorings. Once a refactoring has been performed, the locked names are converted back to Java names, inserting qualifiers where necessary to force the original binding. While this procedure does not always succeed (there may be cases for which no suitable qualification exists, or an entity involved in a qualification may be inaccessible), it has proven well-suited for a language such as Java with its intricate problems of hiding, shadowing, and obscuring [12], which are hard to deal with otherwise. However, it is far less useful for a language like Eiffel, in which scopes cannot be nested and the only possibility to resolve name conflicts is by renaming.

While all of the above works focus on implementing refactoring tools, JunGL [32] is a refactoring scripting language providing the necessary infrastructure for such implementations. Like [18], it builds on a graph representation of programs, but adds powerful querying facilities based on a combination of functional and logic programming (Datalog). However, the mechanics of the refactorings must still be specified imperatively, limiting declarativeness of the approach to precondition checking.

Constraint languages (which are inherently declarative) have so far mostly been devised for other problems than refactoring. For instance, CCEL was designed as a constraint expression language for C++ allowing the definition of design, implementation, and stylistic rules that programmers should obey [20]. A CCEL constraint consists of a set of universally quantified variables and an assertion that uses the variables. Constraints are applied to programs by binding the variables to matching com-

ponents of the program, and by evaluating the constraints for each found binding, with a violated constraint indicating a violated rule. A similar approach is taken by more recent work on pluggable type systems [1], which also uses a constraint language (called JavaCOP). While bearing many similarities to the constraint rules that constraint-based refactoring rests on, neither CCEL nor JavaCOP have notions of allowing a program to change in such a way that its components meet the required constraints — this however is a necessary feature of every refactoring language.

3 A Generalized Framework of Constraint-Based Refactoring

A common tenet of all constraint-based refactorings is that for the purpose of a specific refactoring, a program is sufficiently represented by a set of constraint variables and a set of constraints that relate their values, where both are derived from the program to be refactored by application of so-called constraint rules [4, 9, 26, 29, 30]. Depending on the concrete refactoring, the constraint variables represent properties of program elements such as types [29], type parameters [4, 9, 15], or declared accessibility [26], but generally, there is no reason to restrict constraint variables to one kind of program properties per refactoring, just as there is no general limit as to which properties can at all be represented by constraint variables. This observation is the starting point of our conception of a refactoring constraint language.

3.1 Program Elements, Kinds, and Properties

For our purposes, it is sufficient to assume that a program consists of

- a set D of *declared entities* (elsewhere also referred to as *definitions*),
- a set R of *references* to declared entities (also called *uses* of definitions), and
- relationships on subsets of $D \cup R$.

We refer to declared entities and references collectively as *program elements*.

Program elements come in different *kinds*. Kinds that will be found in most object-oriented programming languages are *Class*, *Field*, *Method*, and *Local* for classes, fields, methods, and locals (formal parameters and local variables), respectively, and *Reference* for references to (uses of) declared entities; but generally, different languages may have different kinds of program elements.

Depending on the programming language, different kinds of program elements have different *properties* such as identifiers, types, declaring classes, etc. Deviating from prior work on constraint-based refactoring, which considered only a single property each (and therefore could afford to use brackets to denote types [29] or accessibilities [26]), we do not limit the kind of properties considered by any single refactoring (and therefore use Greek letters to denote properties, writing $e.\pi$ for the property π of an element e). As will be seen, it is useful to organize the kinds of program elements in a subsumption hierarchy and to let subkinds inherit the properties of their superkinds.

It is important to note that a refactoring will usually alter properties of program elements and not the program elements themselves — the elements' identity is always preserved under refactoring (meaning that an element cannot become another one; in

fact, it is legitimate to think of program elements as objects in the object-oriented sense, as well as of kinds as classes, and of properties as fields). It is the properties, and not the program elements, that are mapped to the constraint variables of a constraint-based refactoring.

3.2 Domains

Each *property* is associated with a *domain* from which its possible *values* are drawn. Although the domains of properties are essentially the domains of constraint variables most of which will be represented by the set of integers (see below), at the refactoring constraint language level it is useful to distinguish different domains of properties. For instance, the domain of a property *identifier* (ι), *Identifier*, is the set of all valid identifiers of a programming language, and the domain of the property *accessibility* (α) in Java is the set {*private*, *package*, *protected*, *public*}. Domains are essentially types, and constraints are typed; a constraint such as $r.\iota = d.\alpha$, although solvable by the solver (if the integer representations of the corresponding domains overlap), does not make sense.

Program-dependent domains Some domains' members are drawn from the elements of a program. For instance, the value of a *location* property that associates a method with its declaring class is a class, which is also program element (of kind *Class*). Since properties are constraint variables, and since standard constraint solvers know nothing of program elements as defined here, we cannot use kinds as the domains of properties. Instead, we allow domains of properties to be based on kinds, by introducing an injective mapping $[\cdot]$ from kinds to domains, writing $[K]$ for the domain corresponding to kind K and (by extending the mapping from kinds to their elements) $[e]$ for the value corresponding to program element $e \in K$.

Ordered Program-Dependent Domains Program-dependent domains such as [*Class*] may be (partially) ordered: in this case, the ordering relation has to be extracted from the program from which the domain is drawn. In the example of [*Class*], one ordering relation is the subclass relation; another would be the nesting of classes (if allowed by the language).

3.3 Constraints, Constraint Rules, and Constraint-Based Refactoring

The constraint variables and constraints that represent a refactoring problem are generated from the program to be refactored by application of so-called *constraint rules*. Each constraint rule is of the general form

$$\frac{\textit{program queries}}{\textit{constraints}}$$

where the rule precedent is a logical expression (see below) selecting the program elements for which the rule is to take effect, and the rule consequent lists the constraints to be generated. The program queries contain variables (which are not constraint variables!) that are bound to program elements during application of the rule;

the properties of these program elements make the constraint variables of the constraints generated by the application. Each rule is implicitly universally quantified over its variables representing program elements, and application of the rule to a program will find all combinations of program elements matching the queries expressed in the rule precedent. For instance, if the query *binds*(*r*, *d*) of the rule

$$\frac{binds(r,\, d)}{r.\iota = d.\iota \quad r.\tau = d.\tau}$$

finds the pairs (r_1, d_1) and (r_2, d_2), the constraints $r_1.\iota = d_1.\iota$, $r_1.\tau = d_1.\tau$, $r_2.\iota = d_2.\iota$, and $r_2.\tau = d_2.\tau$ will be generated and added to the constraint set (note how the rule involves both identifier, ι, and type, τ, properties). Adjacent expressions above and below the bar are implicitly conjoined; explicit logical junctions are also possible (examples will be given in the following sections).

A constraint system generated from a program by application of the constraint rules is always solved with the variable values reflecting the program as is (the *initial values*); it may however be invalidated by assigning one or more variables new values to reflect the goal of the refactoring. Solving the invalidated constraint system then amounts to computing the additional changes mandated by the refactoring. Note how this blurs the usual distinction between precondition checking and the mechanics of a refactoring: an unsatisfiable constraint system reflects a precondition violation.

3.4 Program Queries and Writing Back Solutions

The program queries constituting the precedents of the constraint rules are basically Datalog-like [2, 32] expressions whose predicates are matched against a (thought) fact base representing the program to be refactored. Every predicate has a name and a number of variables as arguments; variables occurring in more than one predicate refer to the same program element. Evaluating these queries produces all tuples of program elements satisfying the rule precedent. How the queries are evaluated is outside the scope of this paper; in practice, queries will be transformed to searches of the AST of the queried program, but generating an intermediate representation of the program and storing it in a database to speed up querying is also feasible.

Once the constraints have been generated and solved, the variable assignments that constitute the solution must be translated to necessary changes of the program to be refactored, a process that we refer to as *writing back the solution*. Depending on the property a constraint variable represents, writing back amounts to changing a component of a declaration (such as identifier, type, accessibility, or other modifiers) or a change of location (declaring class) of a program element, or it may lead to the introduction or deletion of code (a novel aspect that will be detailed in Section 4.4).

3.5 Specifying a Refactoring

If the constraint rules identified for a given programming language are sufficient to guarantee meaning preservation for all possible changes of the constrained program properties, every solution of the constraint system generated from a given program corresponds to a refactoring of that program. Specification of a concrete refactoring such as RENAME or GENERALIZE DECLARED TYPE therefore amounts to narrowing the

solution space to the solutions reflecting the changes associated with that refactoring, that is, the *refactoring intent*.

Specification of a concrete constraint-based refactoring usually involves specifying

- the kinds of program elements the refactoring can be applied to (e.g., all declared entities in the case of RENAME, or typed entities in the case of GENERALIZE DECLARED TYPE),
- the properties that are to be changed by the refactoring (e.g., identifiers in the case of RENAME, and declared types in the case of GENERALIZE DECLARED TYPE), as well as other properties whose adjustment may make a concrete refactoring possible (such as the identifiers of other program elements, e.g. references), and
- to which (sets of) values the changeable properties may be changed by the refactoring (e.g., a supertype in the case of GENERALIZE DECLARED TYPE).

All this information is to be supplied by the *author of the refactoring*. A concrete application of the refactoring further involves specification of

- the concrete program element(s) to which the refactoring is to be applied and
- the target values of the specified element(s)'(s) properties.

This information is to be supplied by the *user of the refactoring*.

4 Challenges of Constraint-Based Refactoring

While the previous section identified the basic constituents of a constraint-based refactoring framework, this section deals with specific problems that we encountered during our work on implementing constraint-based refactoring tools using this framework. The nature of these problems and their repeated occurrence in various refactorings that we have worked on gave rise to the definition of REFACOLA as will be presented in Section 5. One such problem, the *handling of foresight* (necessary for MOVE refactorings that change the AST of a program) requires techniques whose presentation exceeds the spatial limits of this paper; it is presented in a companion paper [27].

4.1 Indirection

Specifying the semantics of programming languages as constraints occasionally involves indirection. For instance, in Java for a member declared *package local* to be accessible by a reference r on receiver q, the type t of q must be defined in the same package the reference is located in, as would be expressed by the constraint $t.\pi = r.\pi$ (in which π denotes the *package* property). However, if the type t of the receiver q is variable (because a refactoring such as GENERALIZE DECLARED TYPE [29] is allowed to change it), this constraint does not model the problem adequately, since t, which is bound to a program element during application of a constraint rule (cf. Section 3.3), during constraint solution always denotes the same type, no matter whether the type of q, $q.\tau$, is changed by a refactoring. What we would really need to express is

$$q.\tau.\pi = r.\pi \tag{1}$$

in which $q.\tau$ denotes the property representing the type of q, and $q.\tau.\pi$ denotes the property representing the package of the value of $q.\tau$. Note that which property $q.\tau.\pi$

denotes depends on the value of $q.\tau$, which is itself a property (and thus variable). Thus, the package property of the receiver type is indirectly accessed.

As detailed in Section 3.2, the members of the domain of a property cannot themselves have properties, so that indirections of the above kind cannot be expressed directly. However, with the aid of program-dependent domains (here: [*Type*] as the domain of $q.\tau$), the indirection of (1) can be expressed as a quantified constraint [27]

$$\exists\, t \in \mathit{Type}\colon q.\tau = [t] \wedge t.\pi = r.\pi \tag{2}$$

which is satisfied only if there is a program element t that is located in the same package as r and whose corresponding program-dependent domain value $[t]$ is the value of $q.\tau$. It turns out that (2) directly maps to the *element* constraint [17, 31] offered by many constraint solvers: if i and x are integer variables and A is an array of integer variables, $\mathit{element}(i, A, x)$ is satisfied if and only if the constraint $A[i] = x$ is satisfied. For instance, using *element* the indirection constraint (1) can be expressed as

$$\mathit{element}(q.\tau, \Pi, r.\pi)$$

in which Π is an array of package properties such that $\Pi[i]$ is $t_i.\pi$, the package property of type t_i whose corresponding value $[t_i]$ is encoded (through the mapping of the domain [*Type*] to the set of integers; cf. Section 3.2) by integer i, and in which $q.\tau$. (whose domain is [*Type*]) serves as the index into Π.

4.2 Reducing the Solution Space: Generating the Necessary Constraints Only

The usual approach to constraint-based refactoring is to apply all constraint rules to all elements of a program to be refactored, thereby generating huge numbers of constraints, including ones for program elements that are not involved in an intended refactoring. Depending on how the generated constraint system is solved, this procedure may not only waste resources in constraint generation, it can also produce myriads of solutions the user of the refactoring did not request, or has no interest in.

As stated in Section 3.5, input to the concrete application of a refactoring are the variable assignments that reflect the refactoring intent, and the set of other variables (properties) whose value may be changed by the refactoring. Of the latter, only the variables participating in constraints that may be invalidated by the former are of interest; all other variable assignments represent solutions that are independent from the refactoring intent, and should therefore not be generated. An algorithm that determines which constraints and constraint variables are to be generated to constrain precisely these properties is shown in Figure 1.

The algorithm is straightforward. It starts with the set of properties whose value must change, determines all constraints derivable from the program that directly constrain their new values, and replaces the occurrences of the properties in the constraints with constants reflecting the new values. For all other properties constrained by these constraints, it checks for each one whether it may be changed by the refactoring: if not, it replaces its occurrences in the constraints with the property's initial value (again a constant); otherwise, it applies the procedure recursively to the property, i.e., it determines which other constraints constrain the property (thereby indirectly constraining the properties to be changed) and so forth. All constraints deter-

Algorithm *GenerateConstraints*
Input:
M, the set of properties whose values must change
N, the set of properties whose values may change
v_0, a function mapping properties to their initial values
v_f, a function mapping the members of M to their new values
R, a set of constraint rules
Output:
C, a set of constraints
Steps:

```
1.  let P = M, Q = ∅, C = ∅
2.  repeat
3.    move one property p from P to Q
4.    for each constraint rule r in R
5.      if r can constrain p
6.        for each constraint c generated by r, constraining p
7.          for each property p' occurring in c
8.            if p' ∈ M
9.              replace every occurrence of p' in c with v_f(p')
10.           else if p' ∉ N
11.             replace every occurrence of p' in c with v_0(p')
12.           else if p' ∉ Q
13.             add p' to P
14.         add c to C
15. until P = ∅
```

Figure 1: Algorithm computing the set of constraints to be generated for a specific refactoring.

mined by this algorithm constrain, either directly or indirectly (through properties that may be changed), the properties that must be changed, and are therefore necessary; all others can be dispensed with. The constraint variables generated by the algorithm are those constrained by its generated constraints.

4.3 Determining the Best Solution: Soft Constraints

Even with the algorithm of Figure 1 in place, a constraint system generated from a refactoring problem has usually more than one solution. The first solution produced by a general purpose constraint solver depends on internals of the solver, and may not be the one that a custom algorithm tailored for the specific problem computes (that is, the "best" solution).

To find a best solution, contemporary solvers allow the definition of an objective variable (also called a soft constraint) whose value they will then maximize or minimize. This can be exploited for refactoring problems, namely by defining an objective variable as accumulating *penalties* for violating certain aptness criteria to be respected by a good solution, such as changing as few properties as possible or preferring insertion of a cast at the call site of an overloaded method over renaming it (see below). In general, penalties can be expressed as equality constraints of the form

$$c_i = \text{if } \mathit{constraint}(\pi_i, \ldots) \text{ then } p_i \text{ else } 0$$

in which c_i is a constraint variable representing the cost of property π_i having a certain value (which one being specified by $\mathit{constraint}(\pi_i, \ldots)$, which may involve other

properties including the original value of π_i) and in which p_i is an integer constant representing the penalty. The solver must then be instructed to minimize an objective variable c whose value is constrained to the sum of all c_i.

Soft constraints come at a high price, however: in the worst case, finding the minimum requires exploration of the complete solution space, which is generally exponential in the number of constraint variables. Therefore, specifying penalties must be carefully traded against local search strategies that try to solve a constraint system starting from a solution that was invalidated by changing one or more variable values.

4.4 Defaults and Introductions

Sometimes the solution of a refactoring problem requires the introduction of a new program element. For instance, in the Java program

```
class Server {
  boolean m(String s) { return s.length() > 0; }
  void m(Comparable s) {…}
}
class Client {{ new a.Server().m("abc"); }}
```

the parameter type of `Server.m(String)`, `String`, can be generalized to `CharSequence` without affecting the method's well-typedness, allowing instances of other implementations of `CharSequence` to be passed. However, this will make the client's invocation of `m` ambiguous unless a cast to `CharSequence` is inserted before the actual parameter `"abc"`. Since a constraint solver cannot introduce new program elements, we have to assume that such a cast, c, is already present in the program, only that it is not visible in the program text, because the cast's target type property, $c.\tau$, has a default value, namely the type of the expression to be cast (here: `String`). Should the solver assign such a "hidden" property a different value (different than the default), this value will materialize in the program text once the solution of the constraint system is written back (Section 3.4). The sparing use of non-default values (i.e., use only if the refactoring is impossible otherwise, or requires even less apt changes) can be enforced by corresponding soft constraints (penalties; see above), in the above example by penalizing $c.\tau \neq r.\tau$, in which r represents the reference to be cast by the (hidden) cast c.[1]

4.5 Miscellaneous

To save space, we omit a number of other challenges and their solutions here. This includes the handling of external program elements such as those contained in libraries (which may constrain a refactoring, but may not themselves be subject to change), the reduction of large domains such as *Identifier* or [*Class*] to smaller ones that do not contain values leading to equivalent solutions (used for RENAME FEATURE in Section

[1] Note that inserting a cast is not the only possibility to fix the presented refactoring problem: other constraints of the program permitting, accessibility of the method competing in the binding, `Server.m(Comparable)`, could be lowered to `private`, or one of the two methods could be renamed. It is one of the strengths of our approach that if constraints for all three properties are generated, the refactoring will consider all possible solutions, and will choose the one imposing the least cost (in terms of the penalties defined).

6.1), and the modelling of orderings that are themselves subject to change by a refactoring (such as class and type hierarchies for refactorings like EXTRACT INTERFACE [29] or REPLACE INHERITANCE WITH DELEGATION [14], or nesting of program elements for MOVE [8] refactorings). However, we can assure the reader that all these challenges can be solved by suitable encodings of the involved domains.

5 The Refactoring Constraint Language REFACOLA

Based on the generalized framework presented in Section 3 and on our solutions to the challenges identified in Section 4 we have devised REFACOLA, a *refa*ctoring *co*nstraint *la*nguage allowing us to express the constraint rules specific to a programming language, as well as to specify refactorings relying on these rules, in a declarative way. Examples of specifications in REFACOLA can be found in the figures below; because of a lack of space, we do not present its syntax rules here. More information on REFACOLA can be found under http://www.feu.de/ps/prjs/refacola.

5.1 The REFACOLA Language and Framework

As can be seen in Figure 2, the REFACOLA language has three kinds of modules: languages, rulesets, and refactorings. A *language module* consists of the definitions of kinds (of program elements; cf. Section 3.1), their properties, domains, and the signatures of program queries. The kinds are arranged in a subkind hierarchy (using <:) and the properties associated with each kind (if any; appended to it in curly braces) are inherited by its subkinds. Properties are associated with domains (separated by a colon), which are either predefined (such as Identifier or Boolean) or enumerated (such as Accessibility in Java or C#) or program-dependent (such as [Class]; cf. Section 3.2). Properties with the same domain are compatible: for instance, class and type properties can be used interchangeably if the domain of both is [Class].[2] The (partial) ordering of program-dependent domains is specified by referring to a program query extracting the ordering from the program to be refactored. For instance, in Eiffel the domain [Class] has two partial orders, one being defined by a query inherits and the other by a query inherits-non-conforming [6 §8.6.9]. For the first ordering defined, <= and < can be used as relation symbols (see, e.g., Figure 5); for all that follow, the query predicate defining the order has to be used. Queries are defined as signatures only; they provide the interface to the querying component of the REFACOLA framework (see below). ENTITY and REFERENCE are predefined kinds corresponding to *D* and *R*, respectively (cf Section 3.1).

A *ruleset module* imports a language module (using the for keyword) and consists of a set of rules, each carrying an identifier. The specification of a rule is comprised of the declaration of the variables serving as placeholders for program elements (introduced by for all), an if-part for the rule precedent, and a then-part for the rule consequent. The precedents of rules are Datalog-like [2] expressions whose atoms are

[2] Note that, for the sake of simplicity, we ignore parameterized types throughout this paper; that they can be treated with constraints has been shown elsewhere, e.g. in [4, 9, 15].

```
language Eiffel
kinds
   Deferrable <: ENTITY {deferred}
   Class <: Deferrable
   Assignable <: REFERENCE {type}
   Feature <: Deferrable
properties
   deferred : Boolean
   type : Type
domains Type = [Class] ordered by inherits
queries
   has(Class, Feature) "Class defines or redefines or inherits Feature"
   create(Assignable) "object of Assignable's type is created and assigned to it"
   inherits(Class, Class) "1st Class inherits from 2nd"
```

```
ruleset Deferring for Eiffel
rules
   deferred-class
      for all c : Class  f : Feature
      if      has(c, f)
      then    f.deferred = true implies c.deferred = true
   instantiation
      for all a : Assignable
      if      create(a)
      then    a.type.deferred = false
```

```
refactoring ChangeDeferredState for Eiffel uses Deferring
forced changes deferred of DeclaredEntity
allowed changes deferred of Class
```

Figure 2: Sample language, ruleset, and refactoring modules for the deferred property (corresponding to abstract in Java et al.) of classes and features (members) in Eiffel. The rule named instantiation constrains the type of an (variable or attribute) to one that is instantiable (i.e., not tagged as deferred), if the assignable is used in an instance creation expression ([6 §8.20.6]). Note that this rule involves indirection (Section 4.1); this is necessary if the type of an assignable is subject to change by a refactoring (such as CHANGE DECLARED TYPE; see Section 6.3). The ordering of the program-dependent domain Type is defined by the inherits query; the ordering is not used in this example.

program queries of the imported language module; the consequents are sets of constraints that are to be generated for the program elements to which the rules apply. The constraints supported by REFACOLA are (currently) those supported by our used constraint solver (Cream [28]) and include program-dependent orderings (using the Relation constraint [28] with the relations extracted as described above) and indirection (Section 4.1).

A *refactoring module* imports a language and one or more ruleset modules of the same language, and specifies which kind of program element the refactoring is to be applied to, which properties it is to change (the "forced changes"), and also which other properties of which other elements (if any) may be changed in the course of the refactoring (the "allowed changes"; cf. Section 3.5).

Based on a refactoring module and the language and ruleset modules it depends on, the REFACOLA compiler generates code that plugs into the REFACOLA framework that is itself embedded in an IDE such as Eclipse, MonoDevelop, or EiffelStudio (see Figure 3; currently, only Eclipse is supported, with adapters to the ASTs of MonoDevelop and EiffelStudio). The framework provides the implementations of the program queries, the constraint generator (which includes an implementation of the *Generate-Constraints* algorithm of Figure 1), and the routines necessary for writing back the solutions into refactored programs. Currently, small adapters are still hand-crafted to

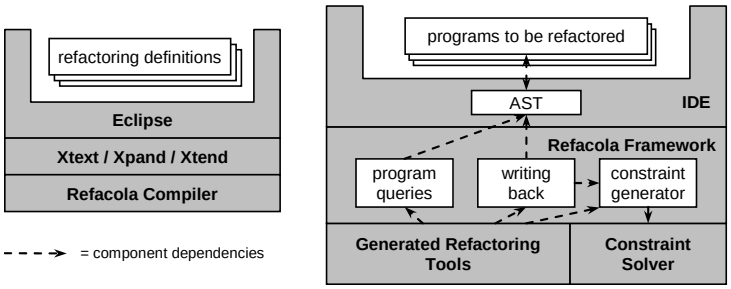

Figure 3: REFACOLA compiler and framework and their embeddings in IDEs.

map the language definitions specified in REFACOLA to the types of the AST. Also, a full-fledged implementation of the framework on the target platform would adapt the refactorings to the IDE's user interface. As of today, however, we have only implemented the batch application required to produce the results presented in Section 7.

5.2 Implementation of the Compiler

Our REFACOLA compiler is implemented using the Xtext [34] language development framework, which is itself Eclipse-based [5] (cf. Figure 3). The necessary type checks (e.g., that properties are only qualified by program elements for which they have been defined, and that constraints and queries are supplied with properties of the required domains) are performed using xtext-typesystem [35]. The code fragments ("plug-ins") that — together with the REFACOLA framework and a constraint solver providing the necessary infrastructure — implement the refactorings specified in REFACOLA are generated using Xpand [33] and are based on templates specific to the programming language and target IDE.

6 Application

To demonstrate the expressiveness of REFACOLA, we first present specifications of the RENAME, CHANGE ACCESSIBILITY, and CHANGE DECLARED TYPE refactorings for the Eiffel programming language, and then explain how they are combined into a single refactoring whose applicability exceeds that of its components. The applicability of the refactorings is evaluated in Section 7.

6.1 The RENAME Refactorings

Eiffel has no explicit namespaces, so that every class must have a unique name. This is expressed as the simple REFACOLA constraint rule (put into the context of a language definition in Figure 4)

```
for all c1, c2 : Class if c1 != c2 then c1.id != c2.id
```

which alone governs the RENAME CLASS refactoring. A similar condition applies to renaming features (members); however, here the situation is more complicated.

```
language Eiffel
kinds
   DeclaredEntity <: ENTITY {id}
   Class <: DeclaredEntity
   Feature <: DeclaredEntity
   Local <: DeclaredEntity
   Renaming <: Feature {inheritedId}
   Reference <: REFERENCE {id}
properties
   id : Identifier
   inheritedId : Identifier
queries
   binds(Reference, DeclaredEntity) "Reference binds to DeclaredEntity"
   has(Class, DeclaredEntity) "Class defines or inherits Feature or Local"
   redefines(Feature, Feature) "1st Feature redefines 2nd"
   renames(Renaming, Feature) "Renaming renames inherited Feature, poss. default"
   merged(Renaming, Renaming) "Renamings are merged in immediate subclass"
   split(Renaming, Renaming) "Renamings are different feat. in immediate subclass"
```

```
ruleset Naming for Eiffel
rules
   binding
      for all r : Reference, d : DeclaredEntity
      if binds(r, d)
      then r.id = d.id
   unique-class-identifier
      for all c1, c2 : Class
      if      c1 != c2
      then    c1.id != c2.id
   unique-feature-identifier-1
      for all c : Class, f1, f2 : Feature
      if      has(c, f1)  has(c, f2)  f1 != f2
      then    f1.id != f2.id
   unique-feature-identifier-2
      for all c : Class, f : Feature, l : Local
      if      has(c, f)  has(c, l)
      then    f.id != l.id
   redefine-feature
      for all f1, f2 : Feature if redefines(f2, f1) then f2.id = f1.id
   rename-feature
      for all f : Feature, r : Renaming if renames(r, f) then r.inheritedId = f.id
   merging
      for all r1, r2 : Renaming if merged(r1, r2) then r1.id = r2.id
   splitting
      for all r1, r2 : Renaming if split(r1, r2) then r1.id != r2.id
```

```
refactoring RenameFeature for Eiffel uses Naming
forced changes
  id of DeclaredEntity
allowed changes
  id of DeclaredEntity values {old, fresh}
  id of Reference
penalties
   changing-identifiers
      for all  d : DeclaredEntity penalize d.id != old d.id with 1
   inserting-rename-clauses
      for all  r : Renaming penalize r.id != r.inheritedId with 2
```

Figure 4: REFACOLA modules for the RENAME FEATURE refactoring for Eiffel.

In Eiffel, a class can *define* a feature, *inherit* it from a superclass, *redefine* an inherited feature (corresponding to overriding), *rename* it (can be combined with redefining), or *undefine* it (i.e., make it abstract). Since Eiffel has no overloading, the names of features available in a class must be unique within that class, and also different from all local variables and formal parameters (together referred to as locals) of that class. In particular, renaming is mandatory if a class inherits two features of the same name from different superclasses, or one that has the same name as one of its own features. The problem is illustrated by the sample program

```
class A feature i : ANY end
class B inherit A feature j : ANY end
class C inherit A rename i as j end end
```

for which renaming feature i of class A to "j" (as a refactoring) must either be rejected (because subclass B already has a feature of the same name), or B.j must be given a fresh name, or a rename clause must be introduced in B which renames the inherited feature. Note that, that A.i is renamed to "j" in C (by a corresponding rename clause) does not constrain the refactoring, even if this means that after the refactoring, the feature is renamed to the same name — instead, in the above example, rather than updating the reference i in the rename clause of C to j, the rename clause can be dropped, giving us the refactored program

```
class A feature j : ANY end
class B inherit A rename j as i end feature j : ANY end
class C inherit A end
```

Alternatively, as mentioned above, feature B.j could have been renamed by the refactoring, saving the renaming of the inherited j.

The REFACOLA definitions for a RENAME refactoring that incorporates enough of the Eiffel specification to handle all options are shown in Figure 4. Except for the handling of rename clauses and the special treatment of repeated inheritance [9 §8.16.2], all definitions are straightforward.

To handle rename clauses as described above, we introduce a special kind of feature, called Renaming, and assume default renamings (defaulting to the same name; cf. Section 4.4) for all inherited features that are not explicitly renamed. This implies that the query binds(r, d) must bind r to a default renaming if the target of r inherits the feature without explicitly renaming it. Renaming declares an additional property, inheritedID, for the identifier of the feature it renames (which may itself be a renaming, default or proper); this is used to keep track of the identifier of that feature (the *inherited identifier* [6 §8.6.18]), which is necessary for determining the default status of a renaming upon writing back the solution of the constraint system (only renamings r for which r.id ≠ r.inheritedId translate to rename clauses in the program; cf. Section 4.4), and also for expressing the penalty inserting-rename-clauses, suggesting that renamings should not needlessly be introduced by the solver (cf. Section 4.3).

In case of repeated inheritance, i.e., inheritance of a feature from two immediate superclasses that have in turn inherited the feature from a common ancestor (the "diamond problem" of multiple inheritance), Eiffel merges the two inherited features into one, unless one or both are renamed in the superclasses so that their names differ — in that case, the subclass inherits two (different) features (the Repeated Inheritance rule; [9 §8.16.2]). Renaming of repeatedly inherited features may thus cause the splitting of a feature into two, or the merging of two features into one, which both likely affect the meaning of the program. Therefore, renaming must maintain the merging status of repeatedly inherited features, which is guaranteed by the rules named merging and splitting.

6.2 The CHANGE ACCESSIBILITY Refactoring

The classes of an Eiffel program do not only all exist in the same global namespace, they are also accessible ("available" in Eiffel jargon) by all other classes of the same program. The accessibility of the features of a class is controlled by an explicit export

mechanism: each feature definition can have a set of classes attached from whose bodies (including the bodies of the classes' subclasses) the feature is accessible. By way of re-exporting using export clauses [6 §8.7], subclasses may change the export status of their inherited features, which includes making them accessible by fewer (including no) classes. This accounts for one half of so-called *catcalls* (where "cat" stands for "change availability or type" [19]), which threaten subtyping.[3] A refactoring should therefore not introduce catcalls.

The following program illustrates some of the constraints controlling accessibility:

```
class A feature {C, D} m do … end end
class B inherit A end
class C feature m local a : A do … a.m … end end
class D feature m local b : B do … b.m … end end
```

In this program, accessibility of A.m could be reduced to include just C, but the access of m on an instance of B from class D would require introduction of an export {D} m clause in class B, as in

```
class A feature {C} m do … end end
class B inherit A export {D} m end end
```

This however is insufficient, since making B.m inaccessible from C (the semantics of the export clause is not additive) makes the call a.m in C a catcall. The export clause in B must therefore include C as well.

The definitions for a CHANGE ACCESSIBILITY refactoring for Eiffel are shown in Figure 5.[4] They are explained as follows:

- While an Eiffel class may introduce one feature clause or export clause per feature defined or re-exported, both a feature clause and an export clause can list several features that then all share the same export status. To express this, feature clauses and export clauses (jointly represented by program elements of kind Export) also have an accessibility property, and the accessibility of features is inferred from that of their exports by constraining them to equal the accessibility of their clause (the accessibility-inference rule). Note that, were the export clause to which a feature belongs subject to change by a refactoring, this constraint would require indirection, i.e., f.accessibility = f.export.accessibility (see Section 4.1), with export being a property of features with program-dependent domain [Export] (see Section 3.2).
- The accessibility property is set valued. This is reflected in the accessible-feature rule, which requires that the location of a reference to a feature, which is (the body of) a class, is a subclass of at least one class in accessibility, the set of classes the feature is exported to. The rule consequent therefore involves an existentially quantified constraint (cf. [27]); its semantics is not detailed here since for technical reasons, in our implementation we will have to replace set-valuedness of accessibility with single-valuedness (see Section 7).
- It is a prerequisite of design by contract that all clients of a feature are able to check that the preconditions of the feature are satisfied (by querying the features

[3] While [19] defines catcalls statically, the Eiffel compiler [7] inserts runtime type checks signalling exceptions on the occurrence of catcalls, so that introducing catcalls does not lead to uncompilable code, but may lead to new runtime errors and therefore may change behaviour.

[4] Note that we use different language and ruleset modules here only to make each refactoring specification self-contained. In practice, modules would be shared between refactorings.

```
language Eiffel
kinds
   Class <: ENTITY
   Export <: DeclaredEntity {accessibility}
   Feature <: DeclaredEntity {accessibility}
   Reference <: REFERENCE {location}
properties
   accessibility : SetOf(Type)
   location : Type
domains
   Type = [Class] ordered by inherits
queries
   feature(Export, Feature) "Feature is a feature of the clause Export"
   accesses(Reference, DeclaredEntity) "Ref. accesses DeclEnt. from other class"
   requires(Feature, Feature) "1st Feature references 2nd Feature in its precond."
   reexports(Feature, Feature) "1st Feature redefines export status of 2nd"
   inherits(Class, Class) "1st Class inherits from 2nd"
```

```
ruleset Exporting for Eiffel
rules
   accessibility-inference
      for all e : Export, f : Feature
      if      feature(e, f)
      then    f.accessibility = e.accessibility
   accessible-feature
      for all r : Reference  f : Feature
      if      accesses(r, f)
      then    exists c : Class ([c] in f.accessibility and r.location <= [c])
   precondition-export
      for all r : Reference, f1, f2 : Feature
      if      accesses(r, f1)  requires(f1, f2)
      then    exists c : Class ([c] in f2.accessibility and r.location <= [c])
   no-catcalls
      for all r : Reference  f1, f2 : Feature
      if      accesses(r, f1)  reexports(f2, f1)
      then    exists c : Class ([c] in f2.accessibility and r.location <= [c])
```

```
refactoring ChangeAccessibility for Eiffel uses Exporting
forced changes accessibility of Feature
allowed changes accessibility of Feature, accessibility of Export
```

Figure 5: REFACOLA modules for the CHANGE ACCESSIBILITY refactoring.

that are used in the precondition) [19, 6 §8.9.5]. This is expressed in precondition-export, which requires that each feature referenced from the precondition of another feature of the same class is accessible to at least the clients of the referencing feature.

- To avoid the introduction of catcalls to a program (cf. above), the no-catcalls rule requires that if a referenced feature is re-exported in a subclass, the re-exported feature of the subclass must also be accessible from the location of the reference.

6.3 The CHANGE DECLARED TYPE Refactoring

Using more general types in the declarations of variables is considered good practice, since it means depending on fewer features, thereby increasing decoupling (the Interface Segregation Principle [16]). However, generalizing the declared type of a variable is constrained by the use of that variable (which features are being accessed on it), and also, via assignment compatibility, by the type of the other variables it gets assigned to (whose types are constrained accordingly). Specializing a declared type is constrained similarly, the differences being that assignments propagate type change in the opposite direction and that the use of the type need not be checked: if the program

to be refactored is free of catcalls, features defined by a supertype that are available to clients are also available — to the same clients — from its subtypes.

The situation is more complicated, however, when the parameter types of redefining or redefined (the Eiffel terms for overriding and overridden) routines (methods) are to be changed: since Eiffel allows covariant redefinition of all parameters, a parameter type of a redefined routine may be generalized, and a parameter type of a redefining routine may be specialized, by a refactoring. However, since covariant redefinition may lead to catcalls (here the call of a routine with a parameter that may not be accepted by a redefining routine in a subclass, threatening static subtyping), care must be taken that no CHANGE DECLARED TYPE refactoring introduces a catcall.

Changing a declared type is different from the previous refactorings in that it changes static binding of references to features, which could affect the meaning of a program. However, in Eiffel (unlike in Java for instance) the binding changes that may be caused by a change of declared type depend on the type of the features' targets (aka receivers) alone (recall that Eiffel has no overloading). Since in Eiffel (again unlike in Java), all feature accesses (including field accesses) are dynamically dispatched, a change of receiver or parameter type never leads to a change of dynamic binding, so that except for creating new objects (instantiation; see below), maintaining (static) type correctness is enough to preserve binding.

The following short sample program illustrates some of the challenges posed by the CHANGE DECLARED TYPE refactoring in Eiffel:

```
class A end
class B inherit A feature n do end end
deferred class C feature m(b1 : B) deferred end end
class D inherit C redefine m end feature m(b2 : B) do b2.n end end
class CLIENT feature m local d:D; b3:B do create(b3); d.m(b3) end end
```

Here, type changes are constrained as follows:

- `b1 : B` can be generalized to `b1 : A`, and `d : D` can be generalized to `d : C`, but not both at the same time, since this would make `d.m(b3)` in class `CLIENT` a catcall statically (cf. Footnote 3);
- `b2 : B` cannot be generalized to `b2 : A` since `A` does not define feature `n`; and
- `b3 : B` cannot be generalized to `b3 : A` since this would require either generalizing `b2` or both `d` and `b1`, where both alternatives, as we have just seen, are prohibited.

As for specialization,

- `b1 : B` cannot be specialized without specializing `b2` with it;
- `b2 : B` cannot be specialized without specializing `b3` with it; and
- `b3 : B` can be specialized to `b3 : D`, but not to `b3 : C`, since this would make the instantiation `create(b3)` illegal (instantiation of a deferred type).

The definitions necessary for the CHANGE DECLARED TYPE refactoring are shown in Figure 6. They are explained as follows:

- `Assignment` is the usual subtyping constraint covering assignment compatibility: it requires that the type of the right-hand side of the assignment is a subtype of the type of the left-hand side.
- `Feature-defined` constrains the type of the receiver (called *target* in Eiffel parlance) of a feature access to descendants of the root definition of that feature (called its *seed*; including the seed itself). Thus, the access is constrained to bind to a *version* ([6 §8.16.8]) of the feature (where versions are defined as having the same seed).

```
language Eiffel
kinds
   DeclaredEntity <: ENTITY {id}
   Variable <: DeclaredEntity {type, location}
   Feature <: Variable
   Formal <: Variable
   Reference <: REFERENCE {id, type, location}
   Class <: DeclaredEntity
properties
   id : Identifier
   type : Type
   location : Type
domains Type = [Class] ordered by inherits
queries
   assignment(Reference, Reference) "2nd Reference is assigned to the 1st"
   binds(Reference, DeclaredEntity) "Reference binds to DeclaredEntity"
   target(Reference, Reference) "2nd Reference is the target (receiver) of 1st"
   seed(Feature, Feature) "2nd Feature is seed (root definition) of 1st Feature"
   defines(Class, Feature) "Feature is defined, redefined, or renamed in Class"
   actual(Reference, Reference, Index) "1st Ref. is actual of 2nd Ref. at Index"
   formal(Formal, Feature, Index) "Formal is formal of Feature at Index"
   redefines(Feature, Feature) "2nd Feature redefines 1st"
   create(Reference) "object of Reference's type is created and assigned to it"
   current(Reference) "Reference is Current"
   result(Reference, Feature) "Reference is Result of query Feature"
   like(Variable, Variable) "1st Variable declared to have the type of 2nd"
   inherits(Class, Class) "1st Class inherits from 2nd"
```

```
ruleset Typing for Eiffel
rules
   assignment
      for all l, r : Reference  if assignment(l, r)  then r.type <= l.type
   feature-defined
      for all r, t : Reference  f, s : Feature
      if      binds(r, f)  target(r, t)  seed(f, s)
      then    t.type <= s.location
   version-binding
      for all r, t : Reference  f1, f2, s : Feature  c : Class
      if      binds(r, f1)  target(r, t)  seed(f1, s)  seed(f2, s)  defines(c, f2)
      then    t.type = [c] implies (r.id = f2.id and r.type = f2.type)
   accessible-feature
      for all r, t : Reference  f1, f2, s : Feature  c : Class
      if      binds(r, f1)  target(r, t)  seed(f1, s)  seed(f2, s)  defines(c, f2)
      then    t.type = [c] implies
               exists c' : Class ([c'] in f2.accessibility and r.location <= [c'])
   routine-call
      for all r,t,a : Reference  r1,r2,s : Feature  c : Class  f : Formal  i:Index
      if      binds(r, r1)  target(r, t)  seed(r1, s)  seed(r2, s)  defines(c, r2)
               actual(a, r, i)  formal(f, r2, i)
      then    t.type = [c] implies a.type <= f.type
   instantiation
      for all r : Reference if create(r) then r.type = old r.type
   covariant-feature-redefinition
      for all f1, f2 : Feature if redefines(f2, f1) then f2.type <= f1.type
   covariant-parameter-redefinition
      for all r1, r2 : Feature  f1, f2 : Formal  i : Index
      if      redefines(r2, r1)  formal(f1, r1, i)  formal(f2, r2, i)
      then    f2.type <= f1.type
   no-catcalls
      for all r : Reference  r1, r2 : Feature  f1, f2 : Formal  i : Index
      if      binds(r, f1)  redefines(r2, r1)  formal(f1, r1, i)  formal(f2, r2,i)
      then    f2.type = f1.type
   type-of-Current
      for all r : Reference if current(r) then r.type = r.location
   type-of-Result
      for all r : Reference  f : Feature if result(r, f) then r.type = f.type
   anchored-type
      for all v1, v2 : Variable if like(v1, v2) then v1.type = v2.type
```

```
refactoring ChangeDeclaredType for Eiffel uses Typing
forced changes type of Variable
allowed changes type of Variable, type of Reference, id of Reference
```

Figure 6: REFACOLA modules for the CHANGE DECLARED TYPE refactoring.

- Version-binding constrains the identifier and type of a reference to equal those of the version of the feature the reference binds to, which depends on the type of the target. Since this type may be changed by the refactoring, this rule compiles a table of conditional constraints (one constraint per version of the feature) of which only one will be active per solution (which one depending on the value of t.type).
- Accessible-feature makes sure that a feature accessed from a changed receiver type is still accessible; it works analogously to version-binding (see below for a discussion).
- Routine-call constrains the types of the actual parameters of a routine call to subtypes of their corresponding formal parameter types (analogously to the assignment rule). The variable i of type Index is implicitly constrained to the number of parameters of the routine.
- Since in Eiffel the class that is instantiated depends on the type of the variable used in the instantiation, and since replacing an object with one of a different class potentially affects behaviour, the instantiation rule prevents the change of declared type of variables used in instantiations. This rule could be relaxed to allow changes to subtypes, if existing contracts guaranteed behavioural subtyping; however, a refactoring tool has no way of checking such contracts. Also, this would require that the class defining the subtype is not deferred (abstract; but note how, if the refactoring definition is extended to include the Deferring ruleset of Figure 2, it could set deferred to false, if no deferred feature of the class prevents this).
- Covariant-feature-redefinition and covariant-parameter-redefinition constrain all types involved in a feature redefinition (in Eiffel, this includes fields) to subtypes.
- No-catcalls requires that redefinition of a referenced routine (method) leaves parameter types unchanged.
- Type-of-Current and type-of-Result constrain the types of the special references representing the current instance (this in Java et al.) and the return value of a function.
- Anchored-type constrains an anchored type to equal the type it is anchored to [6 §8.11.17].

Note that one half of the version-binding rule constrains the identifiers of a reference and the feature that is being referenced, and thus really belongs to the Naming ruleset of Figure 4. In fact, ignoring the type constraints generated by version-binding, it is a generalization of the binding rule of the Naming ruleset, namely one that takes the variability of the type of the reference's target into account (which could be ignored for RENAME FEATURE, since this does not change types). Indeed, as can easily be seen, if t.type is constant, which of the if-parts t.type = [c] of the conditional constraints generated by version-binding will be satisfied is already known when the rule is applied (namely for that application for which [c] equals the current type of the target) — for all that are not (all but a single one), the constraint can be dropped, since it does not constrain the solution (a conditional constraint with a false precedent is always satisfied). Version-binding of Figure 6 thus collapses to the (unconditional) binding rule of Figure 4. Note that the ChangeDeclaredType definition allows the change of identifiers of references; this enables the refactoring in cases in which different versions of a feature have different names (so that references must be updated).

Identifiers are not the only "foreign" property to be handled by CHANGE DECLARED TYPE — by changing the type of a target, a reference using that target may

re-bind to a version of the feature that is inaccessible for the client holding the reference. While accessibility would be covered by the accessible-feature rule of the Exporting ruleset of Figure 5, this rule, like binding of the Naming ruleset, assumes that the (static) binding of a reference to a feature is not changed by a refactoring, which is not the case for CHANGE DECLARED TYPE. Instead, the accessible-feature rule of the Exporting ruleset (Figure 5) must be generalized to that of Typing (Figure 6) which, analogously to the version-binding rule, introduces a table of constraints.

6.4 Combining Several Refactorings into One

While for maximum applicability both identifiers and accessibilities must be handled by CHANGE DECLARED TYPE, changing them is not part of the primary refactoring intent. On the other hand, RENAME FEATURE, CHANGE ACCESSIBILITY, and CHANGE DECLARED TYPE are naturally combined into a single CHANGE FEATURE DECLARATION refactoring allowing the intended change of identifier, accessibility, and one or more parameter types of a routine, all in one step. For this to work correctly, the effect of one constituent refactoring must be immediately visible by the others, which is achieved by letting the constraints generated by each ruleset share those constraint variables that represent same properties of same program elements. Also, a property assumed as variable in one ruleset must not be assumed as constant in another (cf. the above adaptations necessitated by changeability of declared type). Both is achieved by merging the language definitions (cf. Section 5.1) underlying the different ruleset and refactoring definitions into one.

Generally, combining the specifications of different elementary refactorings can increase the applicability of each individual refactoring, if the refactoring is allowed to make additional changes controlled by the other refactoring specifications. For instance, given the program

```
class A feature {ANY} f do end end
class B inherit A export {C} f end end
class C feature g local a : A; b : B do create b; a := b; a.f end end
```

using the ruleset of CHANGE ACCESSIBILITY alone it is not possible to change accessibility of the field A.f to {NONE}. By combining the rulesets of CHANGE ACCESSIBILITY and CHANGE DECLARED TYPE, however, this presents no problem: since the rules of the Typing ruleset (including the adapted accessible-feature rule) allow it that the declared type of the local a in class C is changed to B, and since there are no calls of f on a supertype of A in this program that could trigger Exporting rule no-catcalls on A.f, the refactoring can be performed (and the constraint solver finds the solution). However, changing the declared type of a local such as a in the above example is normally *not* associated with a CHANGE ACCESSIBILITY refactoring (or a CHANGE FEATURE DECLARATION refactoring for that matter); in practice, such a *generalized refactoring*, which blurs the boundaries between individual refactorings, would require some advanced interaction facilities with the user (who might not understand why the program is refactored the way it is), including an explanation component such as [13].

7 Experimental Results

To demonstrate the adequacy of REFACOLA and the above refactoring specifications, and also to measure the efficacy of our *GenerateConstraints* algorithm, we have systematically (i.e., in batch mode) applied each refactoring to all suitable program elements (as specified in the refactoring definitions) of several Eiffel programs. The programs and indicators of their size are listed in Table 1; all programs are taken from the Gobo Eiffel Tools and Libraries [10], and are subsystems of EiffelStudio [7].

The refactorings were applied as follows. RENAME FEATURE was applied to all feature definitions with "other" as the target name. This name is frequently used in the benchmark programs (160 times), thus causing a fair number of name conflicts. CHANGE ACCESSIBILITY was also applied to all feature definitions, with target NONE (corresponding to private). CHANGE DECLARED TYPE was applied to features whose declared type was defined in the same project; targets were all immediate supertypes of the type, if in the same project.

The results of applying the individual refactorings to all programs of Table 1 are shown in Table 2. The numbers were obtained making a couple of restrictions:

- Since renaming is largely constrained by inequality constraints and since the refactoring definition of Figure 4 restricts the domains for identifiers a refactoring may change to two elements (the old and a fresh name), the solution space of renaming is of the order $O(2^n)$ (where n the number of features potentially having a name conflict), which is a problem for finding the best solutions if n is large. Therefore, we extended the *GenerateConstraints* algorithm of Figure 1 with an early evaluation step, pruning constraint generation for constraints known to be always satisfied, which is inherently the case for constraints of the kind `e1.id != e2.id`, if the domains of `e1.id` and `e2.id` are disjoint by construction (cf. Figure 4).
- Because of a limitation of our used constraint solver (Cream [28]), which does not support set-valued domains, we changed the domain of the `accessibility` property to `[Class]`, and adapted the constraint rules accordingly. That this presents no threat to the validity of our results can be seen by observing that only 8 (amounting to 1.2%) of all explicit exports in our sample projects listed more than one class.
- The times for generating the constraint system include querying, but not creating or searching the AST (which was first transformed to a database representation).

The results of Table 2 are interpreted as follows. RENAME FEATURE to "other" was of course always possible (since the domain of each identifier contained a fresh name; see the refactoring definition in Figure 4); that it is also extremely fast is due to the early evaluation, which reduces the search space dramatically (generating only 8 constraint variables on average). Without it, searching for a best solution (with a mini-

Table 1: Sample programs and their sizes.

Program	SLOC	Classes	Features	Program	SLOC	Classes	Features
Gobo Argument	2293	13	158	Gobo Pattern	240	6	11
Gobo Kernel	22913	143	1201	Gobo Regexp	8339	29	378
Gobo Lexical	22139	264	1168	Gobo String	73785	118	2270
Gobo Math	5233	13	331	Gobo Time	5664	34	369
Gobo Parse	18496	69	824	Gobo Utility	7710	46	543
				total	166812	735	7253

Table 2: Results of applying the generated refactorings to the programs of Table 1

Ref.	Ap.[$]	Metric	Avg	Min	¼ Q	Med	¾ Q	Max
Rename Feature	7253 /7253 (100%)	# of Constr. Variables	8	1	2	3	5	1002
		# of Generated Constr.	378	1	67	137	583	2115
		# of Generab. Constr.[†]	1232598	145331	483937	922843	2387543	2387543
		Reduction by ($\log_{10}$)	3.7	2.4	3.3	3.6	4.2	5.7
		Time Generating [ms][§]	336	<.5	125	328	484	984
		Time Solving [ms]	2	<.5	<.5	<.5	<.5	234
Change Accessibil.	5755/7253 (79%)	# of Constr. Variables	7	1	1	1	6	100
		# of Generated Constr.	26	1	3	5	23	1002
		# of Generable Constr.	30781	398	15926	28701	55089	55089
		Reduction by ($\log_{10}$)	3.4	1.2	2.8	3.6	4.3	4.7
		Time Generating [ms]	1	<.5	<.5	<.5	<.5	219
		Time Solving [ms]	54	<.5	<.5	<.5	<.5	19656
Change Decl. Type	3198/9316 (34%)	# of Constr. Variables	9	1	3	4	8	892
		# of Generated Constr.	9	1	2	3	7	1016
		# of Generable Constr.	90150	823	14494	121763	121763	121763
		Reduction by ($\log_{10}$)	4.2	1.1	3.7	4.4	4.6	5.1
		Time Generating [ms]	897	<.5	16	31	62	33296
		Time Solving [ms]	93	<.5	<.5	15	31	15609

[$] applicability: ratio of how often the refactoring could be executed to the number of opportunities for application

[†] that is, generated without using the *GenerateConstraints* algorithm of Figure 1 (cf. Section 4.2)

[§] time measured on a contemporary laptop (2GHz Intel Centrino Duo with 2GB RAM, running Windows 7 operating system)

mum number of renamings; cf. Section 6.1) the solver timed out after 5 seconds in more than half of all cases. Note that the search for a best solution de facto fell prey to early evaluation, causing that no constraint variables are generated for identifier properties that need not change.

Change Accessibility to `NONE` was possible in 79% of all applications of the refactoring. This number may seem surprisingly high, but is explained by the fact that the sample programs are libraries that are not necessarily clients of themselves (so that many features are never referenced; note that this also explains the small numbers of generated constraints). The relatively long maximum solving time of almost 20 s was taken by an unsolvable CSP with 118 constraints; overall, however, there were only 13 applications (all unsolvable) for which the solver required more than 5 s. If the generated CSP was solvable, it had precisely one solution; this follows from the refactoring specification (cf. Figure 5).

Change Declared Type to a supertype was possible in 34% of all cases, indicating that there is opportunity for generalization in the samples (and that having the refactoring at one's disposal is indeed worthwhile). Compared to the other refactorings, times for generating are long; this due to the relatively high number and complexity of the involved constraint rules (Figure 6). Still, total time remains within reasonable bounds most of the times (less than 1 s on average, only 5% of all applications take longer than 5 s), and even the longest times (34 s for generating and solving) are tolerable. The number of generated solutions is 1.2 on average, peaking at 512; however, in only 3% of all applications, there were more than two solutions.

Across all three refactorings, the number of constraints generated using the *GenerateConstraints* algorithm of Figure 1 is dramatically smaller (between 1 and almost 6 orders of magnitude) than what would have been generated without it. Measuring the concomitant reduction in the number of solutions was of course infeasible (due to the exponential size of the solution space) and therefore cannot be reported on here.

To show that the combination of different refactorings can increase applicability of a standard refactoring as suggested in Section 6.4, we have applied CHANGE ACCESSIBILITY, extended with the rulesets of RENAME FEATURE and CHANGE DECLARED TYPE and allowing it to change identifiers and types also, to the sample programs of Table 1 (again with target accessibility `NONE`). This enabled the refactoring in an additional 71 cases, increasing its applicability by 2%.

8 Conclusion

We have argued that for refactoring tools for a given programming language to be correct, they need to incorporate relevant parts of the language's specification. Generalizing prior work on constraint-based refactoring, which has modelled the typing [30] or accessibility [26] rules of Java using constraints, we have devised a refactoring constraint language that, together with accompanying compiler and framework, allows us to generate refactoring tools directly from specifications that mirror the syntactic and semantic rules of the language. By applying generated refactoring tools for the Eiffel programming language to a number of sample programs, we have shown that, despite the generality and declarative nature of our approach, the tools are usable in practice. Another result is that by the combination of the constraint rules underlying different refactorings (representing different aspects of a programming language), we can increase applicability of each individual refactoring.

Acknowledgments

The authors are indebted to Andreas Thies and Erland Müller for their contributions to mastering the refactoring challenges. Emmanuel Stapf helped with the internals of the Eiffel compiler; Naoyuki Tamura pointed us to the *element* constraint.

This work has been made possible by the Deutsche Forschungsgemeinschaft (DFG) under grant STE 906/4-1.

References

1. C Andreae, J Noble, S Markstrum, TD Millstein "A framework for implementing pluggable type systems" in: *Proc. of OOPSLA* (2006) 57–74.
2. S Ceri, G Gottlob, L Tanca "What you always wanted to know about Datalog (and never dared to ask)" *IEEE Trans. Knowl. Data Eng.* 1:1 (1989) 146–166.
3. B Daniel, D Dig, K Garcia, D Marinov "Automated testing of refactoring engines" in: *Proc. of ESEC/SIGSOFT FSE* (2007) 185–194.

4. A Donovan, A Kiezun, MS Tschantz, MD Ernst “Converting Java programs to use generic libraries” in: *Proc. of OOPSLA* (2004) 15–34.
5. *Eclipse* (http://www.eclipse.org)
6. ECMA Standard *Eiffel: Analysis, Design and Programming Language* (ECMA-367, 2nd Edition, 2006).
7. EiffelStudio (http://www.eiffel.com/products/studio)
8. M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).
9. RM Fuhrer, F Tip, A Kiezun, J Dolby, M Keller “Efficiently refactoring Java applications to use generic libraries” in: *Proc. of ECOOP* (2005) 71–96.
10. Gobo Eiffel Tools and Libraries (http://www.gobosoft.com)
11. WG Griswold *Program Restructuring as an Aid to Software Maintenance* (PhD Dissertation, University of Washington, 1992).
12. J Gosling, B Joy, G Steele, G Bracha *The Java Language Specification* (http://java.sun.com/docs/books/jls/).
13. U Junker “QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems" in: *Proc of AAAI* (2004) 167–172.
14. H Kegel, F Steimann “Systematically refactoring inheritance to delegation in Java” in: *Proc. of ICSE* (2008) 431–440.
15. A Kiezun, MD Ernst, F Tip, RM Fuhrer “Refactoring for parameterizing Java classes” in: *Proc. of ICSE* (2007) 437–446.
16. R Martin *Agile Software Development: Principles, Patterns, and Practices* (Prentice Hall, 2003).
17. K Marriott, PJ Stuckey Programming with Constraints (MIT Press, 1998).
18. T Mens, N Van Eetvelde, S Demeyer, D Janssens “Formalizing refactorings with graph transformations” *J. Softw. Maint. Evol.* 17:4 (2005) 247–276.
19. B Meyer *Object-Oriented Software Construction* 2nd edition (Prentice Hall International, 1997).
20. S Meyers, CK Duby, SP Reiss “Constraining the structure and style of object-oriented programs” in: *Proc of PPCP* (1993) 200–209.
21. W Opdyke *Refactoring Object-Oriented Frameworks* (PhD Dissertation, University of Illinois at Urbana-Champaign, 1992).
22. M Schäfer, T Ekman, O de Moor “Sound and extensible renaming for Java” in: *Proc. of OOPSLA* (2008) 277–294.
23. M Schäfer, M Verbaere, T Ekman, O de Moor “Stepping stones over the refactoring Rubicon” in: *Proc. of ECOOP* (2009) 369–393.
24. M Schäfer, J Dolby, M Sridharan, E Torlak, F Tip “Correct refactoring of concurrent Java code” in: *Proc. of ECOOP* (2010) 225–249.
25. M Schäfer, O de Moor “Specifying and implementing refactorings” in: *Proc. of OOPSLA* (2010) 286–301.
26. F Steimann, A Thies “From public to private to absent: Refactoring Java programs under constrained accessibility” in: *Proc. of ECOOP* (2009) 419–443.
27. F Steimann J von Pilgrim “Refactoring with foresight” (http://www.feu.de/ps/docs/Foresight.pdf)
28. N Tamura *Cream: Class Library for Constraint Programming in Java* (http://bach.istc.kobe-u.ac.jp/cream/)
29. F Tip, A Kiezun, D Bäumer “Refactoring for generalization using type constraints” in: *Proc. of OOPSLA* (2003) 13–26.
30. F Tip “Refactoring using type constraints” in: *Proc. of SAS* (2007) 1–17.
31. P Van Hentenryck *Constraint Satisfaction in Logic Programming* (MIT Press 1989).
32. M Verbaere, R Ettinger, O de Moor “JunGL: a scripting language for refactoring” in: *Proc. of ICSE* (2006) 172–181.
33. *Xpand* (http://wiki.eclipse.org/Xpand)
34. *Xtext — Language Development Framework* (http://www.eclipse.org/Xtext)
35. *A Type System Framework for Xtext* (http://eclipselabs.org/p/xtext-typesystem)