2016

Automated Refactoring of Legacy Java Software to Enumerated Types

Raffi T. Khatchadourian
CUNY Hunter College

Automated Software Engineering manuscript No.
(will be inserted by the editor)

Automated Refactoring of Legacy Java Software to Enumerated Types

Raffi Khatchadourian

Received: date / Accepted: date

Abstract Modern Java languages introduce several new features that offer significant improvements over older Java technology. In this article we consider the new `enum` construct, which provides language support for enumerated types. Prior to recent Java languages, programmers needed to employ various patterns (e.g., the weak enum pattern) to compensate for the absence of enumerated types in Java. Unfortunately, these compensation patterns lack several highly-desirable properties of the `enum` construct, most notably, type safety. We present a novel fully-automated approach for transforming legacy Java code to use the new enumeration construct. This semantics-preserving approach increases type safety, produces code that is easier to comprehend, removes unnecessary complexity, and eliminates brittleness problems due to separate compilation. At the core of the proposed approach is an interprocedural type inferencing algorithm which tracks the flow of enumerated values. The algorithm was implemented as an open source, publicly available Eclipse plug-in and evaluated experimentally on 17 large Java benchmarks. Our results indicate that analysis cost is practical and the algorithm can successfully refactor a substantial number of fields to enumerated types. This work is a significant step towards providing automated tool support for migrating legacy Java software to modern Java technologies.

Abstract Keywords software environments · software maintenance · software tools · enumerated types · java · automated refactoring

This material is based upon work supported by the National Science Foundation under grant CCF-0546040.

R. Khatchadourian
Department of Computer Science
Hunter College
City University of New York
New York, NY, 10065 USA
Tel.: +1-212-772-5213
A portion of this work was administered during his time at the Department of Computer Science and Engineering at Ohio State University and at the Department of Computer Systems Technology at New York City College of Technology, City University of New York.
E-mail: raffi.khatchadourian@hunter.cuny.edu

1 Introduction

Modern Java languages introduce a rich set of new features and enhancements such as generics, metadata annotations, boxing/unboxing, and type-safe enumerations (Oracle Corporation 2010). These constructs can ease software development and maintenance and can result in more efficient and robust applications. Even though modern Java languages have backward compatibility with code from previous releases, there are numerous advantages in migrating such legacy code to these new features.

Code migration can be a laborious and expensive task both for code modification and for regression testing. The costs and dangers of migration can be reduced greatly through the use of automated refactoring tools. This article presents a fully-automated semantics-preserving approach for migrating legacy Java code to take advantage of the new type-safe enumeration construct in modern Java.

An *enumerated (enum) type* (Pierce 2002) is a data type whose legal values consist of a fixed, closely related set of items known at compile time (Bloch 2001). Typically, the exact values of the items are not programmatically important: what is significant is that the values are distinct from one another and perhaps ordered in a certain way, hence the term "enumeration." Clearly this is a desirable construct, and since it was not included in the legacy Java languages, developers were forced to use various *compensation patterns* to represent enum types. These patterns produce solutions with varying degrees of uniformity, type safety, expressiveness, and functionality. Of these patterns, the most popular and proclaimed "standard way" (Oracle Corporation 2015) to represent an enumerated type in legacy Java is the *weak enum pattern* (Bloch 2001), also known as *type codes* (Fowler 1999; Kerievsky 2004). This pattern uses declared constants ("codes") defined with relatively small, manually enumerated values. These constants are typically declared as `static final` fields. Variations of this pattern, further discussed in Section 6, include using strings or singleton custom class instances instead of primitive values and pre-compiler directives for languages with a preprocessor. As discussed in Section 2, there are great advantages to migrating compensation patterns in legacy code to proper enum types.

In this article we propose a novel semantics-preserving approach for identifying instances of the weak enum pattern in legacy code and migrating them to the new enum construct. At the core of our approach is an interprocedural type inferencing algorithm which tracks the flow of enumerated values. Given a set of static final fields, the algorithm computes an *enumerization grouping* containing fields, methods, and local variables (including formal parameters) whose types can safely be refactored to use an enum type. The algorithm identifies the fields that are being utilized as enumerated values and all other program entities that are transitively dependent upon these values.

The refactoring approach has been implemented as an Eclipse plug-in. The experimental evaluation used a set of 17 Java programs with a total of 899 thousand lines of code. Our study indicates that (1) the analysis cost is practical, with average running time of 2.48 seconds per thousand lines of code, (2) the weak enum pattern is commonly used in legacy Java software, and (3) the proposed algorithm successfully refactors a large number of static final fields into enumerated types.

This work makes the following specific contributions:

Algorithm design. We present a novel automated refactoring approach for migration to Java enum types. The approach infers which fields are being used as enumerations and identifies all code changes that need to be made in order to introduce the inferred enum types.

Implementation and experimental evaluation. The approach was implemented as an Eclipse plug-in to ensure real-world applicability. A study on 17 Java programs indicates that the proposed techniques are effective and practical. These results advance the state of the art in automated tool support for the evolution of legacy Java code to modern Java technologies.

A shorter version of this work appeared in (Khatchadourian et al 2007), and a demonstration of our preliminary tool, along with implementation details, appeared in (Khatchadourian and Muskalla 2010). In this expanded version, we add the following contributions:

1. Provide complete rules and function definitions of our algorithm. The rules correspond to the complete specification of the Java programming language, which include all contexts where enumerated types may appear. We also fully describe how the approach applies to a running example.
2. Expand our evaluation by providing a complete listing and corresponding explanation of filtered contexts in which we have applied our approach. This gives further insight into the applicability of our proposal on real-world systems.
3. Present a publicly available, open source automated refactoring tool updated for recent Java language versions as a manifestation of our approach.

2 Motivation and Example

An *enumerated type* has values from a fixed set of constants (Bloch 2001). Java has historically provided no language mechanisms for defining enumerated types, leading to the emergence of various compensation patterns. However, the compiler depends on the internal representation (typically `int`) of the symbolically named constants, and type checking can not distinguish between values of the enum type and those of the type internally representing those values.

2.1 Example

Figure 1(a) shows an example in which named constants are used to encode values of enumerated types.[1] For example, field `color` declared at line 7 represents the color which the traffic signal is currently displaying. The values of this field come from the three static final fields `RED`, `YELLOW`, and `GREEN`, which map symbolic names to their associated integer representations. The compile time values of these constants are manually enumerated so that each color can be unambiguously distinguished. Of course, the integer values have no real relationship to the colors they represent. Similarly, field `currentAction` declared at line 19 could take its values from the integer constants in static final fields `IDLE`, `INCREASE_SPEED`, `DECREASE_SPEED`, and `STOP`.

Field `MAX_SPEED` (line 16) defines the maximum speed of the automobile. This field differs from the remaining static final fields: unlike their integer values, which are used only to encode enumerated values, the value of `MAX_SPEED` has a very significant meaning. This key distinction illustrates the difference between fields that are *named constants* (e.g., `MAX_SPEED`) from those participating in the *int enum pattern* (Bloch 2001).[2] In this article we consider

[1] This example was inspired by one of the authors' work at the Center for Automotive Research at the Ohio State University.

[2] A similar pattern called *Type Codes* is described in (Fowler 1999; Kerievsky 2004).

```
1  class TrafficSignal {
2    public static final int RED = 0;
3    public static final int YELLOW = 1;
4    public static final int GREEN = 2;
5    /* Current color of the traffic signal, initially red by
6      default */
7    private int color = RED;
8    /* Accessor for the light's current color */
9    public int getColor() {return this.color;}}
10
11 class Automobile {
12   private static final int IDLE = 0;
13   private static final int INCREASE_SPEED = 1;
14   private static final int DECREASE_SPEED = 2;
15   private static final int STOP = 3;
16   private static final int MAX_SPEED = 140;
17   /* The action this automobile is currently
18     performing, idle by default */
19   private int currentAction = IDLE;
20   /* The current speed of the automobile, initially 5
21     mph. */
22   private int currentSpeed = 5;
23
24   private int react(TrafficSignal signal) {
25     switch(signal.getColor()) {
26       case TrafficSignal.RED: return STOP;
27       case TrafficSignal.YELLOW:
28         // decide whether to stop or go
29         if (this.shouldGo())
30           return INCREASE_SPEED;
31         else return STOP;
32       case TrafficSignal.GREEN: // no change
33         return this.currentAction;
34       default: throw new IllegalArgumentException
35         ("Invalid traffic color");}} // required
36
37   public void drive() {
38     TrafficSignal aSignal = ... ;
39     int reaction = this.react(aSignal);
40     if (reaction != this.currentAction &&
41         (reaction != INCREASE_SPEED ||
42          this.currentSpeed <= MAX_SPEED))
43           this.performAction(reaction);}
44
45   private void performAction(int action) {...}}
```

(a) Using integer constants for enumerated types.

```
1  class TrafficSignal {
2    public enum Color {RED,
3      YELLOW,
4      GREEN};
5    /* Current color of the traffic signal, initially red by
6       default */
7    private Color color = Color.RED;
8    /* Accessor for the light's current color */
9    public Color getColor() {return this.color;}}
10
11 class Automobile {
12   private enum Action {IDLE,
13     INCREASE_SPEED,
14     DECREASE_SPEED,
15     STOP};
16   private static final int MAX_SPEED = 140;
17   /* The action this automobile is currently
18     performing, idle by default */
19   private Action currentAction = Action.IDLE;
20   /* The current speed of the automobile, initially 5
21      mph. */
22   private int currentSpeed = 5;
23
24   private Action react(TrafficSignal signal) {
25     switch(signal.getColor()) {
26       case ~~TrafficSignal~~.RED: return Action.STOP;
27       case ~~TrafficSignal~~.YELLOW:
28         // decide whether to stop or go
29         if (this.shouldGo())
30           return Action.INCREASE_SPEED;
31         else return Action.STOP;
32       case ~~TrafficSignal~~.GREEN: // no change
33         return this.currentAction;
34       default: throw new IllegalArgumentException
35         ("Invalid traffic color");}} // required
36
37   public void drive() {
38     TrafficSignal aSignal = ... ;
39     Action reaction = this.react(aSignal);
40     if (reaction != this.currentAction &&
41         (reaction != Action.INCREASE_SPEED ||
42          this.currentSpeed <= MAX_SPEED))
43           this.performAction(reaction);}
44
45   private void performAction(Action action){...}}
```

(b) Improvements after our refactoring is applied.

Fig. 1 Running example: a hypothetical drive-by-wire application.

a more general version of this pattern which applies to all primitive types[3]; we will refer to it as the *weak enum pattern*. The term "weak" is used to denote the lack of type safety and other features inherent to the pattern.

Figure 1(a) illustrates the use of the weak enum pattern. Clearly, the meaning of `int` depends on the context of where values are used. The programmer is left with the responsibility of *manually* inferring which `int` entities are intended to represent traffic light colors, which are automobile actions, and which are integers. In effect, the programmer would be required to investigate transitive relationships of these program entities to other program entities/-operations. Although the weak enum pattern provides a mechanism to make programmer intent more explicit, it suffers from several significant weaknesses which have been well documented (Bloch 2001; Oracle Corporation 2010).

[3] We exclude boolean from this list for several reasons: (i) The type has only two values, `true` and `false`, thus any transformed enum type can only have two members and (ii) our algorithm becomes simpler due to this exclusion.

2.1.1 Type Safety

The most glaring weakness is the lack of type safety. For example, there is no mechanism to enforce the constraint that `color` gets its values only from the three color fields: any integer value would be acceptable at compile time. Such problems would not be detected until run time, when an exception would be thrown. Perhaps worse, the execution will seem to be normal while behaving in a way not originally intended by the programmer. Problems could also arise from the allowed operations: for example, it would be possible to perform arbitrary integer operations, such as addition or multiplication, upon the color values.

2.1.2 Program Comprehension

The weak enum pattern creates ambiguities at various levels. For example, there are fundamental semantic differences between the constants for automobile actions (beginning on line 12) and `MAX_SPEED` (line 16). Despite these differences, both entities have essentially identical declarations. The programmer depends on documentation and/or extensive interprocedural usage investigation to determine the true intent of the fields. This is also an issue for *multiple* sets of enum constants. For example, methods `getColor` (line 9) and `react` (line 24) declare the same `int` return type, even though the returned entities have very different meaning and context. In essence, the program is less self-documented with respect to the enumerated types, which could have negative effect on software maintenance tasks.

2.1.3 Verbosity

Verbosity and added complexity arises in several areas. First, there is no easy way to print the enumerated values in a meaningful way. Additional code is typically required to produce desirable results, e.g., as in:

```
if (this.color == RED) System.out.println("RED")
```

Second, there is no convenient way to iterate over all values of the enumerated type (Bloch 2001), which requires the developer to manually create such machinery. Third, the weak enum pattern requires the programmer to manually enumerate the values of the constants, which increases the likelihood of errors. For example, different enum constants may be unintentionally assigned the same internal value.

2.1.4 Name spacing

Constant names from different enum types may collide, especially in distributed development environments, as they are not contained in their own name space. For example, the constant `RED` may belong to two enum types defined in the same class. Such a collision would need to be resolved by prefixing the constants with an appropriate identifier (e.g., `COLOR_RED`).

2.1.5 Separate compilation

Finally, the weak enum pattern produces types whose values are brittle (Oracle Corporation 2010). Since the values are compile time constants, at compile time they are inlined into clients. Therefore, if new constants are added in between existing ones, or if the internal

representation of the constants change, clients must be recompiled. Otherwise, the behavior of clients upon referencing the values of the enum type is undefined. Such results are devastating for successful separate compilation.

2.2 Enumerations in Java

The new `enum` construct supports powerful enumerated types that are completely and conveniently type safe, comparable, and serializable; saving the programmer from creating and maintaining verbose custom classes. Enum types increase self-documentation (e.g., a `getColor` method has a return type of `Color`), enable compile-time type checking, allow meaningful printed values, avoid name conflicts, and support separate compilation.

Figure 1(b) shows an *enumerized* version of the running example, in which the static final fields have been replaced by language enumerated types `TrafficSignal.Color` and `Automobile.Action`. The legal values and operations of these new enumerated types are now enforced through compile-time checking. There is a clear distinction between the named constant `MAX_SPEED` and the enumerated values. It is also clear that the result of a call to `react` is an `Action`, which distinguishes it from the return type of `getColor` and makes the API more informative. Programmers are no longer required to enumerate values by hand, or to write extra "pretty printing" code.

After enumerization, the brittleness of the overall system is reduced. For example, suppose we wanted to make `TrafficSignal` compatible with Poland's system, where a yellow and red combination is shown directly after red to alert drivers that a change to green is imminent. After `RED` in Figure 1(a), one could add a new field `RED_YELLOW` with value of 1; the remaining fields' values would have to be incremented. Even if we did not care to modify `Automobile` to accommodate the new color, we would still have to recompile it, since upon the original compilation the constant values for the colors were inlined. In Figure 1(b) additional values can be added easily, and only the enum or the class containing the enum would require recompilation.

3 Enumerization Approach

A refactoring tool which modifies legacy Java code employing the weak enum pattern to utilize the Java `enum` construct faces two major challenges: *inferring enumerated types* and *resolving dependencies*. Inferring enumerated types requires distinguishing between weak enum constants and named constants. Figure 1(a) illustrates this issue through fields `STOP` and `MAX_SPEED`. Although their declarations are very similar, they are conceptually very different: while the value of named constant `MAX_SPEED` is meaningful in integer contexts (e.g., for the integer comparison at line 42), the only requirement on the value of enumerated constant `STOP` is that it should be different from the other integer values representing actions. In general, the uses of the enumerated values are limited to assignments, parameter passing, method return values, and equality comparisons. Named constants are used in a much wider context, including mathematical calculations (e.g., dividing by `java.lang.Math.PI`), various value comparisons (as in line 42), and so on. Determining the category to which a constant field belongs requires investigation of every context in which that field's value is used.

Constant fields are not the only program entities that need to be refactored for enumerization. In Figure 1(a), once it has been inferred that `STOP` is an enumerated constant, we must identify all program entities that also require refactoring due to transitive dependencies

$\mathscr{P}$	original program
$\phi(\mathscr{P})$	$\{f \mid f$ is a static final field of primitive type in $\mathscr{P}\}$
$\mu(\mathscr{P})$	$\{m \mid m$ is a method in $\mathscr{P}\}$
$\upsilon(\mathscr{P})$	$\{l \mid l$ is a variable in $\mathscr{P}\}$
α	variable, field, method
α_{ctxt}	*context* in which α may occur
$\mathscr{P}(ID)$	the program entity corresponding to the terminal identifier expression *ID*

Fig. 2 Formalism notation.

on `STOP`. We say a entity A is *type dependent* on entity B if changing the type of B requires changing the type of A. An example of such a dependency is method `react`: since it returns the integer form of `STOP`, in the refactored version it must return the enum type containing `STOP`. Furthermore, due to the dependence on the return value of `react`, local integer variable `reaction` in `drive` (line 39 in Figure 1(b)) must also be transformed to be of type `Action`.

The next section describes an interprocedural refactoring algorithm which addresses these challenges through careful categorization of the contexts in which migration from the weak enum pattern to the new enum construct is valid. The algorithm identifies all type dependent entities in those contexts, including fields, local variables, method return types, and formal parameters. After all affected entities are identified, they are classified into groups that must share the same enum type. At the end, all automatically transformed code is semantically equivalent to the original.

4 Algorithm

4.1 Assumptions

Our algorithm works on a *closed-world assumption*, meaning that we assume full access to all source code that could possibly affect or be affected by the refactoring. We also assume that we are able to statically identify all references to candidate fields and transitively dependent program entities. This assumption could be invalidated through the use of reflection and custom class loaders.

We also assume that the original source code successfully compiles under a Java ≥ 5 compiler, where enumeration types were first introduced, thus guaranteeing the following properties:

1. There are no uses of the identifier `enum` throughout the program source.[4]
2. The source is type correct.
3. All implicit primitive value conversions are lossless.

Under a Java ≥ 5 compiler, the `enum` identifier is now a reserved keyword and one that would be used in declarations of language enumerated types only. Therefore, assumption (1) allows us to use the `enum` keyword for such purposes only. Assumption (2) is essential as our algorithm is thoroughly dependent on the type relationships of each program entity in the original source. Consequently, the result of our algorithm on type-*incorrect* source is undefined.

[4] Although the focus of our tool is to refactor *legacy* Java software to utilize the new `enum` construct, we do not discriminate against *current* Java software (e.g., those written in Java ≥ 5). In this case, uses of the `enum` identifier for the purpose of declaring language enumerated types is acceptable.

Assumption (3) is also key. Although primitive types do not share many of the same properties as reference types, such as subtype relationships, etc., there exists important relationships between these types that an inferencing algorithm must account for. In fact, this is particularly important to semantic preservation during any transformation of primitive value types to reference types. Similar to the $\leq$ relationship exploited for class type inferencing algorithms in (Palsberg and Schwartzbach 1994), primitive types define *conversion* relationships between them (Gosling et al 2005). Primitives do not enjoy the same polymorphic capabilities that the subtype relationship provides reference types. However, primitives are allowed to be implicitly assigned to values of *different* primitive types much in the same way subtype instances can be assigned to variables of their corresponding supertypes. Such a conversion, in the context of primitives, is called an *implicit widening conversion* (Gosling et al 2005).

Widening conversions, which does not require explicit casts, allows primitive type values to be used interchangeably (through assignment and comparison). Thus, variables of type `double` are allowed to be assigned values of type `int`, `int` variables are allowed to be assigned values of type `char`, and so on. This relationship can be described as `char` $\leq$ `int` $\leq$ `double`. The implicit conversion is legal so long as the value transfer is lossless, that is, no precision of the value is lost by the conversion. Conversions in which precision can be potentially lost are called *narrowing conversions* and must be made explicit through casts. There are, however, exceptions to this rule. For example, a narrowing conversion is allowed to implicitly (i.e., cast-less) take place so as long as the value of the larger[5] type can be resolved to a value that requires less than or equal to the amount of storage allocated for values of smaller types *at compile-time*. For example, although the conversion relationship between `byte` and `int` is `byte` $\leq$ `int`, the `int` constant literal 2, whose value can be vacuously resolved at compile-time, can be stored in a variable of type `byte` without a risk of loss in precision (i.e., lossless). Since our algorithm infers enumerated types by analyzing constants, such a scenario is potentially common and the algorithm must account for this possibility. Seeing that our algorithm assumes that the original program is type-correct, it is safe to further assume that *all* primitive, implicit conversions, widening and narrowing alike, are lossless. Henceforth, transitivity may exist between entities of *different* declared types. Our algorithm does not single out *inter-primitive type* transitivity among program entities. As such, this assumption is necessary to ensure that adequate precision exists in order to preserve semantics.

4.2 Top-level Processing

Procedure *Enumerize*, shown in Figure 3, is the top-level driver of our approach. It takes as input the source code of the original program $\mathscr{P}$, as well as a set $F \subseteq \phi(\mathscr{P})$ of fields (see the notation in Figure 2; parts of this notation were inspired by (Fuhrer et al 2005; Kiezun et al 2007; Steimann et al 2006)). In this article we consider refactoring the "standard" compensation pattern in legacy Java as described in (Bloch 2001; Fowler 1999; Kerievsky 2004; Oracle Corporation 2010). As such, *Enumerize* analyzes only static final fields of primitive types since they may potentially be participating in the weak enum pattern. Function *Enumerizable* (called at line 1) infers which candidate fields are being used as enumerated values and groups them into their corresponding inferred enum types. At line 2, certain semantics-preserving constraints are enforced (further discussed in section 4.6). Fi-

[5] Larger in terms of the maximum capacity held by values of the primitive type in bytes.

procedure $Enumerize(F, \mathscr{P})$
1: $R \leftarrow Enumerizable(F)$
2: $R \leftarrow Unique(R) \cap Distinct(R) \cap Consistent(R)$
3: **for all** $T \in R$ **do**
4: $\quad Transform(T)$
5: **end for**

Fig. 3 Top-level enumerization algorithm.

nally, *Transform* (line 4) performs the actual code refactoring for each inferred enum type T, thus altering the type declarations of each corresponding program entity. The primitive constants are replaced with the new `enum` type declarations. The new `enum` constants are ordered by their original primitive values to enforce a natural ordering, thereby preserving comparability semantics.

4.3 Type Inferencing

Function *Enumerizable*, shown in Figure 4, is at the heart of the proposed approach. This type inferencing algorithm is based on a family of type inferencing approaches from (Palsberg and Schwartzbach 1994), and has two goals:

(i) infer fields that are being used as part of enumerated types (i.e., participating in the weak enum pattern)
(ii) construct minimal sets such that members of the same set must share the same enum type after refactoring

The output of the algorithm is a set of *enumerization sets* containing fields, method declarations, and local variables (including formal parameters) and their minimal groupings that are enumerizable with respect to the input constants.

The algorithm uses a worklist W which is initialized with all given constant fields, as well as a set N of entities that are not amenable to enumerization. A union-find data structure maintains sets of related entities; initially, each input constant field belongs to a separate singleton set. Each worklist element α is a program entity whose type may have to be changed to an enum type. A helper function *Contexts* identifies all contexts (explained next) in which α and its related entities α' appear in $\mathscr{P}$ such that each context α_{ctxt} needs to be examined later in the algorithm.

$Contexts(\alpha, \mathscr{P})$, depicted in Figure 5, includes all *inner-most* (i.e., identifier terminals in the grammar) expressions corresponding to α (excluding those appearing in initializations of constant fields). Furthermore, if α is a method, this set of contexts also includes $Contexts(\alpha', \mathscr{P})$ for every method α' which overrides α or is overridden by α. Similarly, if α is a formal parameter, the set of contexts includes $Contexts(\alpha', \mathscr{P})$ for every corresponding formal parameter α' in an overriding or overridden method. Entities α' need to be considered due to polymorphism. For example, if the return type of a method m is changed from `int` to an enum type, this change must be propagated to all methods overriding m or being overridden by m. Similar propagation is necessary when m's formal parameters are changed (otherwise, method overriding would incorrectly be transformed to method overloading). We denote these sets of dependent entities as *method hierarchies* and *parameter hierarchies*, respectively.

Function *isEnumerizableContext* examines a context α_{ctxt} to determine if it is amenable to enumerization with respect to α by using two helper functions *EnumerizableAscender*

function *Enumerizable*(C)
1: $W \leftarrow C$ /* seed the worklist with the input constants */
2: $N \leftarrow \varnothing$ /* the non-enumerizable set list, initially empty */
3: **for all** $c \in C$ **do**
4: *MakeSet*(c) /* init the union-find data structure */
5: **end for**
6: **while** $W \neq \varnothing$ **do**
7: /* remove an element from the worklist */
8: $\alpha \leftarrow e \mid e \in W$
9: $W \leftarrow W \setminus \{\alpha\}$
10: **for all** $\alpha_{ctxt} \in Contexts(\alpha, \mathscr{P})$ **do**
11: **if** $\neg isEnumerizableContext(\alpha, \alpha_{ctxt})$ **then**
12: /* add to the non-enumerizable list */
13: $N \leftarrow N \cup \{\alpha\}$
14: **break**
15: **end if**
16: /* extract entities to be enumerized due to α */
17: **for all** $\hat{\alpha} \in Extract(\alpha, \alpha_{ctxt})$ **do**
18: **if** $Find(\hat{\alpha}) = \varnothing$ **then**
19: *MakeSet*($\hat{\alpha}$)
20: $W \leftarrow W \cup \{\hat{\alpha}\}$
21: **end if**
22: *Union*($Find(\alpha), Find(\hat{\alpha})$)
23: **end for**
24: **end for**
25: **end while**
26: $F \leftarrow AllSets()$ /* the sets to be returned */
27: **for all** $\alpha' \in N$ **do**
28: $F \leftarrow F \setminus Find(\alpha')$ /* remove nonenum sets */
29: **end for**
30: **return** F /* all sets minus the non-enumerizable sets */

Fig. 4 Building enumerization sets.

$$Contexts(\alpha, \mathscr{P}) = \text{all } \textit{inner-most} \text{ expressions containing } \alpha \cup \begin{cases} \varnothing & \text{if } \alpha \text{ is a local variable or a field} \\ \bigcup_{\alpha' \in MH(\alpha)} Contexts(\alpha', \mathscr{P}) & \text{if } \alpha \text{ is a method} \\ \bigcup_{\alpha' \in PH(\alpha)} Contexts(\alpha', \mathscr{P}) & \text{if } \alpha \text{ is a formal parameter} \end{cases}$$

Fig. 5 Contexts for a program entity α; *MH*/*PH* is the method/parameter hierarchy.

and *EnumerizableDescender*. Upon application, these helper functions examine the context sent to *isEnumerizableContext* by traversing, in disparate directions, the syntax tree of the input expression. The intent of these functions are loosely analogous to that of synthesized and inherited attributes of attribute grammars (Knuth 1967), respectively. Function *Extract* is responsible for determining further transitive relationships due to the enumerization of α. *Extract* also has two helper functions *ExtractionAscender* and *ExtractionDescender* which are similar in flavor to the aforementioned helper functions. For conciseness, in the following discussion we will use the abbreviations *EC*, *EA*, *ED*, *EX*, *XA*, and *XD* to refer to these functions. Essentially, *isEnumerizableContext* and *Extract* serve as canonical names for their intended purpose. *EC* has two parameters: the entity α whose enumerizability is under question and a context α_{ctxt} which is type dependent on α. *EX*, on the other hand, has one parameter α_{ctxt} whose constituent, type dependent program entities must be examined for enumerization.

function $EC(\alpha, \alpha_{ctxt})$
1: **return** $EA(\alpha, \alpha_{ctxt})$
end function

(a) isEnumerizableContext predicate.

function $EX(\alpha_{ctxt})$
1: **return** $XA(\alpha_{ctxt})$
end function

(b) Extraction function.

Fig. 6 Top-level inferencing algorithms.

Function *EC*, portrayed in Figure 6(a), immediately calls *EA* passing it α_{ctxt}, the context to be examined and α, the entity whose enumerization is under question. Figure 7 portrays many of the rules of *EA* which are inductively defined in the grammar. *EA* begins at α_{ctxt} (e.g., *ID*) and *climbs* (or ascends) its way up the grammar until it reaches a *significant ancestor* of α. We say that a statement or expression is a significant ancestor of α if the value of α can be exploited at that point. The ascent is performed via the *Parent* function which returns the parent expression above α_{ctxt} in the syntax tree. The function *contains* helps determine which expression *EA* ascended from.

On the way to the significant ancestor, *EA* may find expressions that are not amenable to enumerization. In that case, *EA* will return *false* and *EC*, in turn, will return the result of *EA*. Such a situation is depicted in the rule for array access/creation in Figure 7. On the other hand, once *EA* successfully reaches the significant ancestor, it will then call *ED* in order to commence a descent down the *pivotal* expression(s); that is, an expression that is consequently type dependent. Much of the rules of *ED* are given in Figure 8. As shown, *ED* completes its descent at the leaf nodes of the syntax tree, returning *true* for terminal IDs and *false* for contexts which are not amenable to enumerization (e.g., literals). *EA* will then, in turn, return the result of *ED*.

4.4 Enumerizable Contexts

EC returns *false* if the given context α_{ctxt} is definitively not enumerizable with respect to α (e.g., α being used as an array index). Otherwise, *EC* returns *true* if α_{ctxt} is *promising* with respect to α–that is, enumerizing α does not adversely affect the relation between α and the enclosing expressions of α_{ctxt}. We say that such a situation is "promising" as opposed to "definite" because there may exist other program entities $\hat{\alpha}$ that are type dependent on α and we cannot yet ensure that every context $\hat{\alpha}_{ctxt}$ in which $\hat{\alpha}$ appears is enumerizable. This additional checking for $\hat{\alpha}$ is performed by *EX*, which extracts the type dependent entities that require further investigation to determine if they are enumerizable with respect to a particular α. The *EX* function is depicted in Figure 6(b) and its helper functions, *XA* and *XD*, are depicted in Figures 9 and 10, respectively. These extracted entities will be put on the worklist and eventually checked by *EC*.

To illustrate the type checking component mechanics we show the application of the *EC* function at each significant ancestor discovered during the evaluation the assignment `color=RED` from line 7 of our motivating example depicted in Figure 1(a). The terminal expression `RED` within the assignment expression `color=RED` would have been returned by *Contexts* when α is `RED`. Applying *EC* for this context we have:

Identifiers
function $EA(\alpha,\text{ID})$
1: **return** $EA(\alpha, Parent(\text{ID}))$

Parenthesized expressions
function $EA(\alpha,(\text{ID}))$
1: **return** $EA(\alpha, Parent(\text{ID}))$

Cast expressions
function $EA(\alpha,(\text{TYPE})\text{EXP})$
1: **return false**

Field access expressions
function $EA(\alpha,\text{EXP.ID})$
1: **return** $EA(\alpha, Parent(\text{EXP}))$

Assignment expressions
function $EA(\alpha,\text{EXP}_1 = \text{EXP}_2)$
1: **return** $ED(\text{EXP}_1) \wedge ED(\text{EXP}_2)$

Subtract assignment expressions
function $EA(\alpha,\text{EXP}_1 \text{ -= } \text{EXP}_2)$
1: **return false**

Divide assignment expressions
function $EA(\alpha,\text{EXP}_1 \text{ /= } \text{EXP}_2)$
1: **return false**

Infix addition expressions
function $EA(\alpha,\text{EXP}_1 + \text{EXP}_2)$
1: **return false**

Infix multiplication expressions
function $EA(\alpha,\text{EXP}_1 * \text{EXP}_2)$
1: **return false**

Prefix unary minus expressions
function $EA(\alpha,\text{-EXP})$
1: **return false**

Postfix increment expressions
function $EA(\alpha,\text{EXP++})$
1: **return false**

Equality expressions
function $EA(\alpha,\text{EXP}_1 == \text{EXP}_2)$
1: **return** $ED(\text{EXP}_1) \wedge ED(\text{EXP}_2)$

Inequality expressions
function $EA(\alpha,\text{EXP}_1 \text{ != } \text{EXP}_2)$
1: **return** $ED(\text{EXP}_1) \wedge ED(\text{EXP}_2)$

Switch statements
function $EA(\alpha, switch(\text{EXP}))$
1: **let** se $=$ $ED(\text{EXP})$
2: **let** ce $=$ **true**
3: **for all** case $\text{EXP}_c \in cases(\text{switch}(\text{EXP}))$ **do**
4: ce $\leftarrow$ ce $\wedge$ $ED(\text{EXP}_c)$
5: **end for**
6: **return** se $\wedge$ ce

Switch case statements
function $EA(\alpha,\text{case EXP})$
1: **return** $EA(\alpha, switchStmt(\text{case EXP}))$

Conditional expressions
function $EA(\alpha,\text{EXP}_1 \text{ ? } \text{EXP}_2 : \text{EXP}_3)$
1: **if** $contains(\text{EXP}_2,\alpha) \vee contains(\text{EXP}_3,\alpha)$ **then**
2: **return** $EA(\alpha, Parent(\text{EXP}_1 \text{ ? } \text{EXP}_2 : \text{EXP}_3))$
3: **else**
4: **return true**
5: **end if**

Array access/creation expressions
function $EA(\alpha,\text{EXP}_1[\text{EXP}_2])$
1: **if** $contains(\text{EXP}_2,\alpha)$ **then**
2: **return false**
3: **else**
4: **return** $EA(\alpha, Parent(\text{EXP}_1))$
5: **end if**

Array initialization expressions
function $EA(\alpha,\{\text{EXP}_1,\ldots,\text{EXP}_n\})$
1: **let** ie $=$ **true**
2: **for** $\text{EXP}_i, 1 \leq i \leq n$ **do**
3: ie $\leftarrow$ ie $\wedge$ $ED(\text{EXP}_i)$
4: **end for**
5: **return** ie

Return statements
function $EA(\alpha,\text{return EXP})$
1: **return true**

Method declaration statements
function $EA(\alpha,\text{ID}(\text{P}_1,\ldots,\text{P}_n))$
1: **let** re $=$ **true**
2: **for all** return $\text{EXP}_r \in returnStmts(\text{ID}(\text{P}_1,\ldots,\text{P}_n))$ **do**
3: re $\leftarrow$ re $\wedge$ $ED(\text{EXP}_r)$
4: **end for**
5: **return** re

Formal parameters
function $EA(\alpha,\text{P}_i)$
1: **let** ae $=$ **true**
2: */*check the ith argument of each invocation of the declaring method*/*
3: **let** $\hat{\alpha} = MethodDecl(\text{P}_i)$
4: **for all** $\hat{\alpha}_{ctxt} \in Invocations(\hat{\alpha}, \mathscr{P})$ **do**
5: ae $\leftarrow$ ae $\wedge$ $ED(Arg(\hat{\alpha}_{ctxt}, i))$
6: **end for**
7: **return** ae

Method invocation expressions
function $EA(\alpha,\text{ID}(\text{EXP}_1,\ldots,\text{EXP}_n))$
1: **for** $\text{EXP}_i, 1 \leq i \leq n$ **do**
2: **if** $contains(\text{EXP}_i,\alpha)$ **then**
3: **return true**
4: **end if**
5: **end for**
6: **return** $EA(\alpha, Parent(\text{ID}(\text{EXP}_1,\ldots,\text{EXP}_n)))$

General statements
function $XA(\alpha,\text{SMT})$
1: **let** se $=$ **true**
2: **for** $\text{EXP} \in Children(\text{SMT})$ **do**
3: se $\leftarrow$ se $\vee ED(\text{EXP})$
4: **end for**
5: **return** se

Fig. 7 Enumerizable ascender.

Integer literals
function *ED*(IL)
1: **return false**

Identifiers
function *ED*(ID)
1: **return true**

Parenthesized expressions
function *ED*((EXP))
1: **return** *ED*(EXP)

Cast expressions
function *ED*((TYPE)EXP)
1: **return false**

Field access expressions
function *ED*(EXP.ID)
1: **return** *ED*(ID)

Assignment expressions
function *ED*(EXP$_1$ = EXP$_2$)
1: **return** *ED*(EXP$_1$) $\wedge$ *ED*(EXP$_2$)

Subtract assignment expressions
function *ED*(EXP$_1$ -= EXP$_2$)
1: **return false**

Divide assignment expressions
function *ED*(EXP$_1$ /= EXP$_2$)
1: **return false**

Infix addition expressions
function *ED*(EXP$_1$ + EXP$_2$)
1: **return false**

Infix multiplication expressions
function *ED*(EXP$_1$ * EXP$_2$)
1: **return false**

Prefix unary minus expressions
function *ED*(-EXP)
1: **return false**

Postfix increment expressions
function *ED*(EXP++)
1: **return false**

Conditional expressions
function *ED*(EXP$_1$? EXP$_2$: EXP$_3$)
1: **return** *ED*(EXP$_2$) $\wedge$ *ED*(EXP$_3$)

Array access expressions
function *ED*(EXP$_1$[EXP$_2$])
1: **return** *ED*(EXP$_1$)

Array creation expressions
function *ED*(TYPE[EXP]{EXP$_1$, ..., EXP$_n$})

1: **return** *ED*({EXP$_1$, ..., EXP$_n$})

Array initialization expressions
function *ED*({EXP$_1$, ..., EXP$_n$})
1: **let** ie $\equiv$ **true**
2: **for** EXP$_i$, $1 \leq i \leq n$ **do**
3: ie $\leftarrow$ ie $\wedge$ *ED*(EXP$_i$)
4: **end for**
5: **return** ie

Method invocation expressions
function *ED*(ID(EXP$_1$, ..., EXP$_n$))

1: **return true**

Fig. 8 Enumerizable descender.

$$
\begin{aligned}
& EC(\texttt{RED},\texttt{RED}) \\
& EA(\texttt{RED},\texttt{RED}) \\
& EA(\texttt{RED},\texttt{color} = \texttt{RED}) && \text{(identifiers)} \\
& ED(\texttt{color}) \wedge ED(\texttt{RED}) && \text{(assignment)} \\
& \mathit{true} \wedge \mathit{true} \equiv \mathit{true} && \text{(identifiers)}
\end{aligned}
$$

As a result, this expression is considered "promising". The subsequent application of *EX* would extract the program entity `color` so that all of its contexts may be checked. Demonstrating this derivation using the rules in Figures 9 and 10, we have:

$$
\begin{aligned}
& EX(\texttt{RED},\texttt{RED}) \\
& XA(\texttt{RED},\texttt{RED}) && \text{(identifiers)} \\
& XA(\texttt{RED},\texttt{color} = \texttt{RED}) && \text{(assignment)} \\
& XD(\texttt{color}) \cup XD(\texttt{RED}) && \text{(identifiers)} \\
& \{\mathscr{P}(\texttt{color})\} \cup \{\mathscr{P}(\texttt{RED})\} \\
& \{\mathscr{P}(\texttt{color}), \mathscr{P}(\texttt{RED})\}
\end{aligned}
$$

where $\mathscr{P}(\texttt{color})$ denotes the program entity corresponding the terminal identifier expression `color` (see Figure 2); in this case the field `color` of class `TrafficSignal`. Consequently,

Identifiers
function $XA(\alpha,\text{ID})$
1: **return** $XA(\alpha, Parent(\text{ID}))$

Parenthesized expressions
function $XA(\alpha,(\text{ID}))$
1: **return** $XA(\alpha, Parent(\text{ID}))$

Cast expressions
function $XA(\alpha,(\text{TYPE})\text{EXP})$
1: **return** $\varnothing$

Field access expressions
function $XA(\alpha,\text{EXP.ID})$
1: **return** $XA(\alpha, Parent(\text{EXP}))$

Assignment expressions
function $XA(\alpha,\text{EXP}_1 = \text{EXP}_2)$
1: **return** $XD(\text{EXP}_1) \cup XD(\text{EXP}_2)$

Subtract assignment expressions
function $XA(\alpha,\text{EXP}_1 \text{ -= } \text{EXP}_2)$
1: **return** $\varnothing$

Divide assignment expressions
function $XA(\alpha,\text{EXP}_1 \text{ /= } \text{EXP}_2)$
1: **return** $\varnothing$

Infix addition expressions
function $XA(\alpha,\text{EXP}_1 + \text{EXP}_2)$
1: **return** $\varnothing$

Infix multiplication expressions
function $XA(\alpha,\text{EXP}_1 * \text{EXP}_2)$
1: **return** $\varnothing$

Prefix unary minus expressions
function $XA(\alpha,\text{-EXP})$
1: **return** $\varnothing$

Postfix increment expressions
function $XA(\alpha,\text{EXP++})$
1: **return** $\varnothing$

Equality expressions
function $XA(\alpha,\text{EXP}_1 == \text{EXP}_2)$
1: **return** $XD(\text{EXP}_2) \cup XD(\text{EXP}_1)$

Inequality expressions
function $XA(\alpha,\text{EXP}_1 \text{ != } \text{EXP}_2)$
1: **return** $XD(\text{EXP}_1) \cup XD(\text{EXP}_2)$

Switch statements
function $XA(\alpha, switch(\text{EXP}))$
1: $R \leftarrow XD(\text{EXP})$
2: **for all** case $\text{EXP}_c \in cases(\text{switch(EXP)})$ **do**
3: $R \leftarrow XD(\text{EXP}_c)$
4: **end for**
5: **return** R

Switch case statements
function $XA(\alpha,\text{case EXP})$
1: **return** $XA(\alpha, switchStmt(\text{case EXP}))$

Return statements
function $XA(\alpha,\text{return EXP})$
1: **return** $XD(MethodDecl(\text{return EXP}))$

Conditional expressions
function $XA(\alpha, \text{EXP}_1 \text{ ? } \text{EXP}_2 : \text{EXP}_3)$
1: **if** $contains(\text{EXP}_2,\alpha) \vee contains(\text{EXP}_3,\alpha)$ **then**
2: **return** $XA(\alpha, Parent(\text{EXP}_1 \text{ ? } \text{EXP}_2 : \text{EXP}_3))$
3: **else**
4: **return** $\varnothing$
5: **end if**

Array access/creation expressions
function $XA(\alpha, \text{EXP}_1[\text{EXP}_2])$
1: **if** $contains(\text{EXP}_2,\alpha)$ **then**
2: **return** $\varnothing$
3: **else**
4: **return** $XA(\alpha, Parent(\text{EXP}_1))$
5: **end if**

Array initialization expressions
function $XA(\alpha,\{\text{EXP}_1,\ldots,\text{EXP}_n\})$
1: $R \leftarrow \varnothing$
2: **for** $\text{EXP}_i, 1 \leq i \leq n$ **do**
3: $R \leftarrow R \cup XD(\text{EXP}_i)$
4: **end for**
5: **return** R

Method declaration statements
function $XA(\alpha,\text{ID}(\text{P}_1,\ldots,\text{P}_n))$
1: $R \leftarrow \varnothing$
2: **for all** return $\text{EXP}_r \in returnStmts(\text{ID}(\text{P}_1,\ldots,\text{P}_n))$ **do**
3: $R \leftarrow R \cup XD(\text{EXP}_r)$
4: **end for**
5: **return** R

Formal parameters
function $XA(\alpha,\text{P}_i)$
1: $R \leftarrow \varnothing$
2: */*extract the ith argument of each invocation of the declaring method*/*
3: **let** $\hat{\alpha} = MethodDecl(\text{P}_i)$
4: **for all** $\hat{\alpha}_{ctxt} \in Invocations(\hat{\alpha}, \mathscr{P})$ **do**
5: $R \leftarrow R \cup XD(Arg(\hat{\alpha}_{ctxt}, i))$
6: **end for**
7: **return** R

Method invocation expressions
function $XA(\alpha,\text{ID}(\text{EXP}_1,\ldots,\text{EXP}_n))$
1: $R \leftarrow \varnothing$
2: **for** $\text{EXP}_i, 1 \leq i \leq n$ **do**
3: **if** $contains(\text{EXP}_i,\alpha)$ **then**
4: $R \leftarrow R \cup XD(\text{EXP}_1)$
5: **end if**
6: **end for**
7: **if** $R \neq \varnothing$ **then**
8: **return** R
9: **else**
10: **return** $XA(\alpha, Parent(\text{ID}(\text{EXP}_1,\ldots,\text{EXP}_n)))$
11: **end if**

General statements
function $XA(\alpha,\text{SMT})$
1: $R \leftarrow \varnothing$
2: **for** $\text{EXP} \in Children(\text{SMT})$ **do**
3: $R \leftarrow R \cup XD(\text{EXP})$
4: **end for**
5: **return** R

Fig. 9 Extraction ascender.

Integer literals
function $XD(\text{IL})$
1: **return** $\varnothing$

Identifiers
function $XD(\text{ID})$
1: **return** $\mathscr{P}(\text{ID})$

Parenthesized expressions
function $XD((\text{EXP}))$
1: **return** $XD(\text{EXP})$

Cast expressions
function $XD((\text{TYPE})\text{EXP})$
1: **return** $\varnothing$

Field access expressions
function $XD(\text{EXP.ID})$
1: **return** $XD(\text{ID})$

Assignment expressions
function $XD(\text{EXP}_1 = \text{EXP}_2)$
1: **return** $XD(\text{EXP}_1) \cup XD(\text{EXP}_2)$

Subtract assignment expressions
function $XD(\text{EXP}_1 \text{ -= } \text{EXP}_2)$
1: **return** $\varnothing$

Divide assignment expressions
function $XD(\text{EXP}_1 \text{ /= } \text{EXP}_2)$
1: **return** $\varnothing$

Infix addition expressions
function $XD(\text{EXP}_1 + \text{EXP}_2)$
1: **return** $\varnothing$

Infix multiplication expressions
function $XD(\text{EXP}_1 * \text{EXP}_2)$
1: **return** $\varnothing$

Prefix unary minus expressions
function $XD(\text{-EXP})$
1: **return** $\varnothing$

Postfix increment expressions
function $XD(\text{EXP++})$
1: **return** $\varnothing$

Conditional expressions
function $XD(\text{EXP}_1 \text{ ? } \text{EXP}_2 : \text{EXP}_3)$
1: **return** $XD(\text{EXP}_2) \cup XD(\text{EXP}_3)$

Array access expressions
function $XD(\text{EXP}_1[\text{EXP}_2])$
1: **return** $XD(\text{EXP}_1)$

Array creation expressions
function $XD(\text{TYPE}[\text{EXP}]\{\text{EXP}_1, \ldots, \text{EXP}_n\})$
1: **return** $XD(\{\text{EXP}_1, \ldots, \text{EXP}_n\})$

Array initialization expressions
function $XD(\{\text{EXP}_1, \ldots, \text{EXP}_n\})$
1: $R \leftarrow \varnothing$
2: **for** $\text{EXP}_i, 1 \leq i \leq n$ **do**
3: $R \leftarrow R \cup XD(\text{EXP}_i)$
4: **end for**
5: **return** R

Method invocation expressions
function $XD(\text{ID}(\text{EXP}_1, \ldots, \text{EXP}_n))$
1: **return** $XD(\text{ID})$

Fig. 10 Extraction descender.

this field will make its way to the *Contexts* function via the worklist and the entire process repeats for this entity.

Consider a hypothetical assignment `color=5` when α is `color`; here `color` is type dependent on the integer literal `5`. Using the rules in Figures 7 and 8, we have the following derivation:

$$\begin{aligned}
&EC(\texttt{color}, \texttt{color}) \\
&EA(\texttt{color}, \texttt{color}) \\
&EA(\texttt{color}, \texttt{color} = \texttt{5}) && \text{(identifiers)} \\
&ED(\texttt{color}) \wedge ED(\texttt{5}) && \text{(assignment)} \\
&\mathit{true} \wedge \mathit{false} \equiv \mathit{false} && \text{(identifiers, integer literals)}
\end{aligned}$$

Thus, $EC(\texttt{color}, \texttt{color})$ is determined to be `false`. Because the type of the integer literal cannot be altered to an enum type, `color` also cannot be altered and should be included in set N (line 13 in Figure 4).

There are other situations where type dependencies prevent a program entity from being enumerized. For example, consider the following statement where α is again `RED`: `if(color==arr[RED]) color=GREEN;`. The derivation using our rules would consist of the following:

$$
\begin{array}{rl}
EC(\texttt{DECREASE\_SPEED}, \texttt{DECREASE\_SPEED}) & \\
EA(\texttt{DECREASE\_SPEED}, \texttt{DECREASE\_SPEED}) & \\
EA(\texttt{DECREASE\_SPEED}, \texttt{color} == \texttt{GREEN}\ ?\ \texttt{INCREASE\_SPEED} : \texttt{DECREASE\_SPEED}) & \text{(identifiers)} \\
EA(\texttt{DECREASE\_SPEED}, \texttt{action} = \texttt{color} == \texttt{GREEN}\ ?\ \texttt{INCREASE\_SPEED} : \texttt{DECREASE\_SPEED}) & \text{(conditionals)} \\
ED(\texttt{action}) \wedge ED(\texttt{color} == \texttt{GREEN}\ ?\ \texttt{INCREASE\_SPEED} : \texttt{DECREASE\_SPEED}) & \text{(assignment)} \\
true \wedge ED(\texttt{INCREASE\_SPEED}) \wedge ED(\texttt{DECREASE\_SPEED}) & \text{(ident/conds)} \\
true \wedge true \wedge true \equiv true & \text{(identifiers)}
\end{array}
$$

Fig. 11 *isEnumerizableContext* derivation of `action = color == GREEN ? INCREASE_SPEED : DECREASE_SPEED`.

$$
\begin{array}{rl}
EX(\texttt{DECREASE\_SPEED}, \texttt{DECREASE\_SPEED}) & \\
XA(\texttt{DECREASE\_SPEED}, \texttt{DECREASE\_SPEED}) & \\
XA(\texttt{DECREASE\_SPEED}, \texttt{color} == \texttt{GREEN}\ ?\ \texttt{INCREASE\_SPEED} : \texttt{DECREASE\_SPEED}) & \text{(identifiers)} \\
XA(\texttt{DECREASE\_SPEED}, \texttt{action} = \texttt{color} == \texttt{GREEN}\ ?\ \texttt{INCREASE\_SPEED} : \texttt{DECREASE\_SPEED}) & \text{(conditionals)} \\
XD(\texttt{action}) \cup XD(\texttt{color} == \texttt{GREEN}\ ?\ \texttt{INCREASE\_SPEED} : \texttt{DECREASE\_SPEED}) & \text{(assignment)} \\
\{\mathscr{P}(\texttt{action})\} \cup XD(\texttt{INCREASE\_SPEED}) \cup XD(\texttt{DECREASE\_SPEED}) & \text{(ident/conds)} \\
\{\mathscr{P}(\texttt{action})\} \cup \{\mathscr{P}(\texttt{INCREASE\_SPEED})\} \cup \{\mathscr{P}(\texttt{DECREASE\_SPEED})\} & \text{(identifiers)} \\
\{\mathscr{P}(\texttt{action}), \mathscr{P}(\texttt{INCREASE\_SPEED}), \mathscr{P}(\texttt{DECREASE\_SPEED})\} &
\end{array}
$$

Fig. 12 *Extract* derivation of `action = color == GREEN ? INCREASE_SPEED : DECREASE_SPEED`.

$$
\begin{array}{rl}
EC(\texttt{RED}, \texttt{RED}) & \\
EA(\texttt{RED}, \texttt{RED}) & \\
EA(\texttt{RED}, \texttt{arr}[\texttt{RED}]) & \text{(identifiers)} \\
false & \text{(array access)}
\end{array}
$$

In this case, *EC* returns `false` since it would be impossible to alter the type of `RED` because the index to an array access must be an integral type (Gosling et al 2005). Note that the *then* portion of the if statement is not evaluated as it is not type dependent on α. Although *EX* is not called when *EC* returns *false*, *EX* would nevertheless return $\varnothing$ upon these arguments.

As another example, consider conditional expressions `x?y:z`. Here, we must be careful to distinguish between each expression in which α may (or may not) appear in. If α only appears in EXP_1, we should not check EXP_2 and EXP_3. However, if α appears in either EXP_2 or EXP_3, then both of these expressions must be enumerizable. That is, the entire expression must evaluate to that of an `enum` type in either case (i.e., the *then* or *else* case). Consider the following conditional expression where α is `DECREASE_SPEED`:

```
action = color == GREEN ? INCREASE_SPEED : DECREASE_SPEED
```

Then, we have the derivation as depicted in Figure 11. The extracted set of type dependent entities would be as portrayed in Figure 12.

In general, the enumerizability of particular α may depend on its occurrences within comparison expressions (see the rules for equality/inequality expressions in Figure 7). For

comparison expressions with == and !=, as long as both operand expressions are enumerizable both will be included in the same inferred enum type, and the integer equality/inequality in the original code will be transformed to reference equality/inequality. For <, <=, >, and >=, the refactored code can use the methods from interface `java.lang.Comparable`, which is implemented by all enum types in Java, to preserve comparability semantics amongst the inferred type's members. This holds true so long as the inferred enum type declarations are in the order given by their original primitive representations.

An interesting case is contexts in which polymorphic behavior may occur. In these cases, we need to consider entire hierarchies of program entities. Much of the polymorphic behavior enforcement is implemented with the help of function *Contexts* described earlier, however, additional checks are needed within *isEnumerizableContext* and *Extract* in order to ensure the preservation of program semantics. In particular, the formal parameter expressions and the method invocation expressions require additional investigation of program entities in $\mathscr{P}$. For example, in the case of formal parameters EX must be certain to extract the program entities embedded in the corresponding actual argument expressions for each method invocation in the method hierarchy. These rules are depicted in Figures 9 and 10.

4.5 Transitive Dependencies

In function *Enumerizable*, if either a context which is not amenable to enumerization is encountered, or one that can not be transformed, we mark the set containing the α in question as a "non-enumerizable" set (line 13 in Figure 4). If this is not the case, the algorithm proceeds to extract other program entities that are required to undergo enumerization consideration due to the enumerization of α (line 17). For each of these program entities $\hat{\alpha}$ the following steps are taken. If $\hat{\alpha}$ is not currently contained in an existing set (line 18), which implies that it has not previously been seen, then a new singleton in the union-find data structure is created and consequently added to the worklist (lines 19 and 20). The two sets, the set containing α and the set containing $\hat{\alpha}$, are then merged on line 22 thereby capturing the transitive dependencies between each program entity. Once the computation is complete, i.e., the worklist has emptied, the sets defined implicitly by the union-find data structure are returned minus the non-enumerizable sets (line 30).

Function *Enumerizable* is responsible type inferencing; that is, it ensures that the proposed transformation is type-correct. Its result is a partitioning of program entities, limited to variables, fields, and methods, that are enumerizable with respect to a given set of static final fields. This essential relationship existing between each member of each enumerizable set is expressed by our first *member constraint*, listed as constraint 1 of Figure 13. The constraint simply expresses that all members of each set are enumerizable with respect to the original input constants, of which are also in the set. That is, for all elements k of an enumerizable set K, there exists two sets X and Y such that the element k is a member of the set X, X is a valid partition of the program elements enumerizable in respect to a set of constants Y, and K is a subset of Y. This last clause gives us the flexibility to enumerize only a portion of the original constants if we so desire. The partitioning captures the *minimal* dependency relationships between these entities; if a transformation of one of the elements occurs, then, in order for preserve type correctness, a transformation of *all* elements in its set must also occur. However, we must make further, more subtle considerations as to which sets can be transformed. We discuss such considerations next.

Let $Enumerizable : \mathcal{P}[\phi(\mathscr{P})] \longrightarrow \mathcal{P}^{(2)}[\phi(\mathscr{P}) \cup \mu(\mathscr{P}) \cup \upsilon(\mathscr{P})]$ be a function mapping a set of primitive, static, final fields to a set of *minimal* program entity sets that are enumerizable in respect to those fields. Then, we define:

$$Partitionable(K) \equiv \forall k \in K[\exists X, Y \mid k \in X \wedge X \in Enumerizable(Y) \wedge K \subseteq Y] \tag{1}$$

Let $\iota : \phi(\mathscr{P}) \rightarrow \Sigma^*$ be a function mapping a field to its unqualified identifier. Then, we define:

$$Unique(K) \equiv \forall k_i, k_j \in K[i \neq j \Rightarrow \iota(k_i) \neq \iota(k_j)] \tag{2}$$

Let $\mathbb{P}$ be the set of all legal primitive values and $\sigma : \phi(\mathscr{P}) \rightarrow \mathbb{P}$ be a function mapping a constant to its primitive value. Then, we define:

$$Distinct(K) \equiv \forall k_i, k_j \in K[i \neq j \Rightarrow \sigma(k_i) \neq \sigma(k_j)] \tag{3}$$

Let $\mathbb{V} = \{public, protected, private, package\}$ be the set of legal visibilities and $\vartheta : \phi(\mathscr{P}) \rightarrow \mathbb{V}$ be a function mapping a constant to its visibility. Then, we define:

$$Consistent(K) \equiv \forall k_i, k_j \in K[\vartheta(k_i) = \vartheta(k_j)] \tag{4}$$

Fig. 13 Member constraints for transforming a group of candidate fields K.

Fig. 14 Initial enumeration sets.

4.6 Semantics-preserving Constraints

In additional to analyzing the *usage* of potential enumerated type constants, in order to preserve semantics upon transformation, it is also necessary to analyze their *declarations*. Returning to the *Enumerize* function listed in Figure 3, the functions invoked on line 2 enforce program behavioral preservation by excluding sets containing constants that do not meet the remaining member constraints given in Figure 13. Invocation of the function *Unique* corresponds to the enforcement of constraint 2, *Distinct* to constraint 3, and *Consistent* to constraint 4. Essentially, these constraints express that, for each set to be transformed into a corresponding enum type and for semantics to be preserved, each static final field must be uniquely named (since constants may have originated from different classes), distinctly valued (so that each originally assigned primitive value will correspond to a single memory reference), and consistently visible (since the new enum types are not allowed to have instances with independent visibilities). The resulting intersection of the sets abiding to each of the member constraints is then assigned back to R. At line 4, each set $T \in R$ corresponds to the program entities that will be transformed to the new language enumeration type T and the transformation takes place $\forall T \in R$.

4.7 Application

We now briefly demonstrate how our algorithm would apply to the example code snippet given earlier in Figure 1. A schematic depicting the results of the *Enumerizable* function application appear in Figures 14 and 15. The figures informally represent “snapshots” of

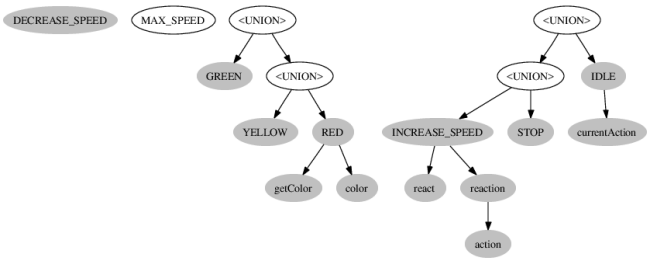

Fig. 15 Enumeration sets produced by our algorithm.

the state of the union-find disjoint data structure at the beginning and the end of the algorithm, respectively. Union-find data structures may be internally represented as trees and the schematic reflects this notion. There are two different types of nodes, valued and unvalued. Valued nodes represent an element (i.e., a field, method, or variable) and are used in producing the output of the algorithm. Unvalued nodes consist of `<UNION>` nodes which serve as logical placeholders marking points in which the sets were merged. Edges connect nodes belonging to the same set. Edge directions depict the order in which nodes were discovered during execution of the algorithm. Edge directions do not necessarily denote transitivity in a particular direction; transitive relationships in respect to enumerization are bidirectional.

In Figure 14, the initial input elements are used to seed the enumeration sets and the application of *Enumerizable* grows the sets as seen in Figure 15. During growth, the sets may be combined due to transitivity of equality/inequality comparisons on both left- and right-hand sides and/or assignments on the right-hand side. The resulting sets depicted in Figure 15 shows that 4 of the original 8 sets have been merged. Sets containing shaded elements designate *enumerizable* sets, that is, sets that contain all elements whose usages are amenable to enumerization. Sets not shaded signify sets that contain at least one element not amenable to enumerization, such as the set containing the element `MAX_SPEED`.

Note that these sets portray the *minimal* dependency information among their elements, therefore, they may be further merged but not split. Also notice that the results produced by our algorithm as applied to the drive-by-wire example are not entirely desirable. Specifically, the automobile action `DECREASE_SPEED` is contained in a different set than that of the other automobile actions due to the current transitive nature of the elements. Surely, when performing the language enumeration type transformation, we desire that all automobile action be grouped together in the same language enumeration type. We leave examining how a result can be automatically suggested by our refactoring for future work, possibly leveraging heuristic techniques from Gravley and Lakhotia (1996).

4.8 Other Patterns

While the weak enum pattern is the proclaimed standard way to represent enumerated types in Java < 5, there are variations of this pattern that can also be used. While these are discussed in more detail in Section 6, we briefly illustrate some instances here. For example, strings instead of primitive values can be used to represent the values:

```
class TrafficSignal {
  public static final String RED = "RED";
  public static final String YELLOW = "YELLOW";
```

```
  // ...
}
```

Or, a custom class can be used:

```
class Color {
    String stringRep;
    private Color(String stringRep) {this.stringRep = stringRep;}
    public static final Color RED = Color("RED");
    public static final Color YELLOW = Color("YELLOW");
    // ...
}
```

In both cases, the reference value, instead of a primitive value, is used to uniquely distinguish the constant values. The benefit of using a custom class, which uses a private constructor, is increased type safety but requires a field to store the constant's string representation.

To deal with such cases, our algorithm would need to be adjusted to work with reference types. There may also be more complex cases with custom classes that involve inheritance. Processing reference types alleviates some of the algorithmic complexity as there are fewer operations for such values (e.g., $<$ would be invalid), yet it would introduce other complexities stemming from casts, etc. We plan to explore analyzing pattern variations in the future by possibly utilizing type constraints (Palsberg and Schwartzbach 1994; Tip et al 2003).

5 Experimental Study

5.1 Implementation

We implemented our algorithm as a open source, publicly available plug-in to the popular Eclipse IDE. Eclipse ASTs with source symbol bindings were used as an intermediate program representation. The plug-in is built over an existing refactoring framework (Bäumer et al 2001) and is coupled with other refactoring support in Eclipse. A special data structure, which extended `LinkedHashSet`, was used to represent the worklist. The data structure includes certain sanity checks for working with array types when elements are added. It also maintains the internal tree data structure that represents the union-find data structure discussed in Section 4.3. Our plug-in is open source and publicly available on GitHub.[6]

To increase applicability to real-world applications, we relaxed the closed-world assumption described in Section 3. For example, if the tool encounters a variable that is transitively dependent upon an element outside of the source code being considered, this variable and all other entities dependent on it are conservatively labeled as non-enumerizable.

Although the tool is currently in a research prototype stage, it contains several useful features, such as a refactoring wizard, before-and-after refactoring preview pane, and AST rewriting. Features remaining to be implemented include a full test suite with regression tests and conformance with other refactoring plug-ins in Eclipse. The tool also has been a proposed project in the Google Summer of Code[7] 2009 and 2010 competitions.

The main development focus has been on producing an intuitive user-interface that provides a mechanism for developers to interactively group constants together to create the new enum type. Figure 16 portrays a screen shot of the current refactoring wizard when applied

[6] `http://github.com/khatchad/Constants-to-Enum-Eclipse-Plugin`

[7] `http://code.google.com/soc`

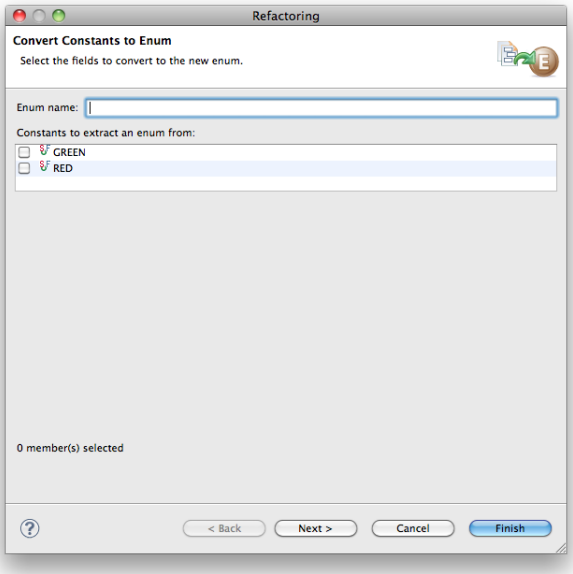

Fig. 16 Screen shot of the enum refactoring wizard.

to the following source code, more precisely, when selecting the `RED` and `GREEN` constants, and Figure 17 depicts the refactoring preview pane:

Listing 1 A simple example to demonstrate features of the Convert Constants to Enumerated Types Eclipse refactoring plug-in.

```
package p;
public class A {
  public static final int RED = 0;
  public static final int GREEN = 1;
  int x() {
    int a = RED;
    a = GREEN;
    return a;
  }
  int y() {
    return x();
  }
}
```

For future work, we plan to incorporate information visualization to enable developers to see the strength of relationships between the constants so that they may make an informed decision of how they should be grouped into enumerated types. In the current state of the tool, constant are grouped in their minimal type-dependent sets such that if one constant in the set is refactored, all other constants in the same set must also be refactored to the same enumerated type. Further plans include but are not limited to providing undo functionality,

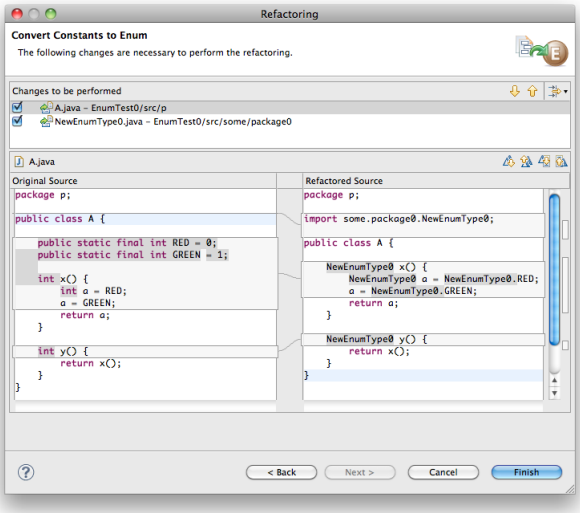

Fig. 17 Screen shot of the enum refactoring preview pane.

refactoring history rollback, and refactoring scripting support. In addition, improvements to the refactoring wizard include conveniently allowing the user to add additional constants to the newly created enum type that were not originally part of the input set. This would better situate CONVERT CONSTANTS TO ENUM to conform to standard refactoring tools in Eclipse.

5.2 Experimental Evaluation

To evaluate the effectiveness of our algorithm, we used the 17 open-source Java applications and libraries listed in Table 1.[8] The second column in the table shows the number of non-blank, non-comment lines of source code, which range from 3K for `jdepend` to 272K for `Azureus`. The third column shows the number of class files after compilation. For each benchmark, the analysis was executed five times on a 2.6 GHz Pentium4 machine with 1 GB RAM. The average running time, in seconds, is shown in column *time* in Table 1. On average, the analysis time was 2.48 seconds per KLOC, which is practical even for large applications.

Column *prim* shows the number of static final fields of primitive types.[9] We separate these fields into two categories. First, certain fields *definitely* cannot be refactored, because the semantics of the program depends on the specific, actual values of these fields. These fields include those that were either directly or transitively dependent on operations that utilized their exact value or created a transitive dependency on an entity which could not be refactored. A complete list of filtered contexts is provided in Table 2. The first column, *Filtered contexts*, displays the contexts which were filtered. The second, *Example Ops*, provides

[8] `java5` denotes the package `java.` included in the Java 5 JDK.

[9] Excludes `boolean` types.

benchmark	KLOC	classes	prim	cands	enum	uses	rtypes	time (s)
ArtOfIllusion	75	378	333	77	77	111	46	207
Azureus	272	1894	1255	399	347	635	173	1269
java5	180	1586	1299	557	450	572	363	760
JavaCup	6	41	55	3	3	3	3	19
jdepend	3	28	13	1	1	1	1	1
JFlex	10	46	140	24	19	27	9	75
JFreeChart	71	420	153	36	24	43	12	128
jGap	6	137	25	4	4	5	1	7
jgraph	14	91	25	6	3	6	1	11
JHotDraw	29	496	34	11	11	24	8	14
junit	8	271	7	2	2	3	1	1
jwps	20	155	156	76	64	102	25	60
sablecc	29	237	16	8	8	10	2	9
tomcat6	153	1164	738	344	335	400	255	346
verbos	5	41	10	6	6	15	2	3
VietPad	11	84	36	17	17	22	4	8
Violet	7	73	36	14	13	20	6	9
Total:	899	7142	4331	1585	1384	1999	915	2227

Table 1 Experimental results.

Filtered Context	Example Ops	Example Uses
Character literal	$==, =>, !=, <, >, <=$	`v == 'c'`
Number literal	$==, =>, !=, <, >, <=$	`v != 28`
Array access	[]	`x[v]`
Array creation	*new int*[],*new double*[]	`new int[v]`
Infix expression	$+, -, /, *, |, \&, <<, >>, +=$	`x = v + s`
Postfix expression	$++, --$	`v++`
Prefix expression	$\sim, ++, --, !, +, -$	`--v`

Table 2 Filtered contexts.

a subset of operations which fall into each context, and column *Example Uses* provides an example of each context–note that the `v` variables in the examples represent constant primitive fields. Since the weak enum pattern only allows the use of literals in the declarations of enumeration fields, other contexts which utilize them were filtered as shown in rows *Character literal* and *Number literal*. Since the semantics of an array access relies on the particular value of the index, any field used as such cannot be refactored. Similarly, fields whose values are utilized in the creation of a new array cannot be refactored (rows *Array access* and *Array creation*). There are a multitude of mathematical operations which, of course, rely on the values of the variables being manipulated. These operations are shown in rows *Infix expression*, *Postfix expression*, and *Prefix expression*. Some of these operations may be valid for certain extensions to weak enum compensation pattern, such as those that employ a bit vectoring over their enumeration values. We also include in this category the fields which cannot be refactored due to lack of access to source code (e.g., a field passed as a parameter to a method defined in a library whose source code is not available).

We categorize the remaining fields to be *candidate fields*. The number of candidate fields per benchmark is shown in column *cands*. The fact that the actual values of these fields do not directly affect the semantics of the program provides a strong indication that they are playing the role of enumerations in the weak enum pattern. The set of candidate fields

along with their corresponding, transitive entities represent the *minimal* set of elements a programmer would have to investigate for refactoring. Note that although these sets are minimal, for three of our benchmarks they still contain well over 300 elements, and several others contain over 50 elements.

The number of fields that our plug-in could safely refactor is shown in column *enum*. The results show that our approach was able to refactor 87% of the fields that could possibly be participating in the weak enum pattern. We randomly sampled approximately 25% of these fields to ensure that they were not named constants or other fields not intended to represent enumerated types. Our sampling showed no such evidence, i.e., the refactored fields we sampled did seem to be participating in the pattern. We discuss possible drawbacks to this analysis in Section 5.3. Moreover, the refactored programs compiled correctly and no additional compiler warnings were issued.

The tool was unable to refactor the remaining 13% of fields because either they or an element of their dependency sets were used in explicit cast expressions. We conservatively choose not to refactor elements used in cast expressions due to the existence of possible side effects on the values of variables though narrowing conversions. For example, consider the following code: `short z = 128; byte x = (byte)z;`. This is a valid cast in Java, but this cast will result in `x` having the value -128 and *not* 128. Clearly, not accounting for such an occurrence prior to refactoring could lead to significant changes in program semantics upon migration. Detecting such changes due to explicit casts is beyond the scope of the work being considered in this article.

Excluding these fields from the refactoring may be problematic if they are truly participating in the weak enum pattern. The problem could perhaps be mitigated through developer intervention, e.g., manually rewriting the code to remove the casts. As our tool reports on each error, the developer, if possible, can remove the cast and rerun the refactoring in these cases.

Of course, fields are not the only program entities whose type requires alteration. Column *uses* shows the total number of declaration sites that must be modified to accommodate the enumerization. The numbers motivate the need for automated tools such as ours. In particular, the large applications require hundreds of code modifications (e.g., over 600 for `Azureus`). These code modifications are spread across many classes and packages, and occur in many distinct methods. Attempting to identify the needed modifications by hand would be a labor-intensive and error-prone task.

Column *rtypes* shows the number of resulting enum types produced by our tool. Note that the number of types is relatively close to the number of enum fields. This indicates that there are few actual enumeration values per enum type, on average about 2.2 per type. This number may not reflect the number of weak enum pattern instances intended by the programmer. Our algorithm is conservative in its type creation, only grouping fields that share transitive dependencies. These are the only fields that *must* share the same enum type upon refactoring. However, given the current state of the program source, dependencies may not exist between all enumerations intended to be grouped as one type. For the running example, `DECREASE_SPEED` should intuitively be grouped with the other vehicle actions. Unfortunately, since it is not currently being referenced by the code, it does not share a dependency with any of the other fields and as a result it is assigned a singleton set (as shown in Figure 15). Clearly, in this case no algorithmic method could guarantee the exact grouping intended by the programmer; however, there are various heuristics that may be employed to better approximate the intended types (e.g., heuristics that take into account lexical proximity of field declarations, similar to what is described in (Gravley and Lakhotia 1996)).

5.3 Threats to Validity

Several threats can undermine the aforementioned evaluation results of our approach. This section considers a number the threats to this study and how we have minimized their effects on the study results.

5.3.1 Internal Validity

In Section 5.2, we explained how we randomly sampled the refactored fields to verify that they were indeed participating in the week enum pattern. However, it may be the case that the original developer has a different intention for these fields than that of what the researcher may derive. This may undermine the sampling affirmation process. To mitigate this threat, the researchers working on this project have over 15 years of Java software development in both industrial and academic environments. As such, there is a high probability that the confirmed fields in the sampling process are accurate. Moreover, the researchers were not involved in the development of any of the subject applications, which reduces the risk of any bias.

5.3.2 External Validity

It may be the case that the selected open-source Java applications and libraries used for study and listed in Table 1 are not representative of Java programs at large. To ensure that a certain level of quality was maintained, we purposefully selected subjects that have been used previously in the literature, including empirical studies. This ensures that the subjects have achieved a particular level of acceptance within the community.

The aforementioned field confirmation sampling rate of approximately 25% may not be large enough to capture the essence of the refactoring's accuracy in only selecting fields participating in the weak enum pattern. To mitigate this, we ensured that a large corpus of open source projects was used so that 25% still encompassed a significant number of sampled fields, totaling $\sim$346.

5.4 Summary

Overall, the experimental results indicate that the analysis cost is practical, that the weak enum pattern is commonly used in legacy Java software, and that the proposed algorithm successfully refactors a large number of fields (and their dependent entities) into enumerated types.

6 Related Work

Fowler (1999); Kerievsky (2004) present the refactoring entitled REPLACE TYPE CODE WITH CLASS. Both detail a series of steps involved in transforming *type codes* (entities subscribing to what we label as the *weak enum pattern* in this article) into instances of custom, type-safe classes utilizing the Singleton pattern (Gamma et al 1995). Bloch (2001) presents a similar solution. While the pattern describes an enum class that seems effective in regards to the same criteria we have presented in this article, the refactoring process is entirely manual and the transformation is not to language enumerated types. Most importantly,

the developer is required to posses *a priori* knowledge of exactly which fields are potentially participating in the type code pattern in order to perform the refactoring. Our proposed approach does not require such knowledge and is completely automated. That is, our approach *infers* in an automated fashion such fields. Furthermore, the developer is presented with the type-dependent groups of the fields which may span multiple classes.

Kumar et al (2012) detail an approach to convert preprocessor macros in C++ programs to C++11 constructs. Although enumeration types may be implemented as macros in C/C++, their approach does not utilize interprocedural type inferencing.

Tip et al (2003) propose two automated refactorings, EXTRACT INTERFACE and PULL UP MEMBERS, both well integrated into the Eclipse IDE. These refactorings deal with *generalizing* Java software in an effort to make it more reusable. Although this proposal shares similar challenges with our approach in respect to precondition checking and interprocedural dependency analysis, there are several key differences. The generalization approach manipulates the interfaces[10] of reference types along with the means in which objects communicate through those interfaces, as our approach entails transforming primitive type entities *to* reference types. Moreover, a method based on *type constraints* (Palsberg and Schwartzbach 1994) is used to resolve dependencies amongst program entities. Sutter et al (2004) also use type constraints in addition to profile information to customize the use of Java library classes. A type constraint approach would have also been conceivable for our work in that similar type constraints may have been formed for primitive types. Nonetheless, a type constraint-based approach for primitive transformation may have proven to be excessive since primitive types do not share many of the same relationships as reference types (e.g., sub-typing). Therefore, we preferred more of a type checking approach as opposed to constraint solving.

Several other approaches (Donovan et al 2004; Fuhrer et al 2005; Tip et al 2004; Kiezun et al 2007; von Dincklage and Diwan 2004; Dig et al 2009) exist to migrate legacy Java source to utilize modern Java language features, in particular generics. Although both generics and language enumeration types serve to improve type safety, the two features are conceptually different and face unique challenges in automated migration. The *instantiation* problem (Fuhrer et al 2005) entails inferring generic type arguments for generic class instances. The *parameterization* problem (Kiezun et al 2007) necessitates inferring generic type parameters for non-generic class declarations. Various challenges include preserving program erasure (Bracha et al 2003), sub-typing compatibility, inferring wild-card types, etc. However, our proposal for inferring *enumerated types*, although not being required to address such issues, must consider other such situations. First, enumerization requires introducing a *new* type in the original source as opposed to introducing a type parameter or argument for an *existing* type. Second, when refactoring primitives one must consider many additional operations that may be invoked on the primitive entities that are not available to reference types. Furthermore, the dependency *flow* must also be taken in account across these operations. For example, in our proposal type dependence not only flows from assignments but also from comparisons.

Steimann et al (2006, 2003) propose an approach to decouple classes with inferred interfaces. Similar to our approach, a new type is introduced in the source (i.e., the inferred interface), and the compile-time types of program entities are altered as a result of the refactoring. Additionally, both approaches do not leverage constraint solving mechanisms, instead, Steimann et al. utilize a static analysis based on (Dean et al 1995). Unlike this pro-

[10] The term *interface* is used here not to solely denote that of Java interfaces, but instead to denote the broader notion of interfaces which, in Java, would also include class declarations.

posed approach, however, our approach must consider more than the transitive closure of assignments beginning on the right-hand side. Again, enumerization entails bidirectional dependencies not only over assignments but also over comparisons.

Automated usage analysis and type inferencing techniques similar to ours also exist for other languages. Eidorff et al (1999) demonstrate a Year 2000 conversion tool utilizing type inferencing techniques for correcting problematic date variables in COBOL (a *weakly-typed* programming language) systems. Ramalingam et al (1999) also exploit usage analysis techniques to identify implicit aggregate structure and programmer intent of COBOL program entities not evident from their declarations.

In fact, proposals for identifying enumerated types exist for COBOL and C. Although our work applies to a significantly different source language, methods for identifying enumerated types in these legacy systems share similar challenges. Deursen and Moonen (1998) present a general approach utilizing judgements and rules for inferring type information from COBOL programs. An in-depth empirical analysis is presented in (Deursen and Moonen 1999). Both COBOL and Java $\leq$ 1.4 do not provide language facilities for enumerated types, and both approaches use a flow-insensitive, interprocedural[11] data flow analysis to discover program entities intended to represent enumerated types. However, our approach is focused more on the *migration* of these entities to a *specific* language enumerated type construct that contains corresponding, preexisting constraints. As a result, our approach must deal with different semantic preservation issues upon transformation, insuring that substitution by the new construct will produce a program with identical behavior upon execution. Moreover, refactoring primitives to reference types presents unique challenges as objects in Java *cannot* share the same memory location; thus, grouping program entities interacting with values from similar literals into corresponding types would not produce an applicable solution. Likewise, our approach must consider modern features such as polymorphism and function overloading during its source analysis and semantic preservation efforts.

Gravley and Lakhotia (1996) tender an approach for identifying enumerated types in C programs that utilize a pre-compiler directive pattern similar to the weak enum pattern that we have described. We see this approach as orthogonal to ours since only the declarations of the constants are analyzed. Furthermore, as mentioned earlier, this approach may be appropriately adapted to enhance the results of our algorithm by leveraging declaration characteristics during grouping.

7 Conclusions and Future Work

In this article we have presented a novel, semantic preserving, type inferencing algorithm which migrates legacy Java code employing the weak enum pattern to instead utilize the modern, type-safe *enum* language construct introduced in modern Java. We implemented our algorithm as a plug-in for the popular Eclipse IDE and evaluated it on 17 open source applications. Our experiments showed that not only did our tool scale well to large applications but was able to refactor 87% of all fields that could possibly be participating in the weak enum pattern. We have publicly distributed our plug-in. In the future, we plan to explore potentially faster intermediate representations of code (e.g., Jimple (Vallée-Rai and et al. 2000)), as well as an enhanced user interface. We also plan to investigate ways of extending our tool to also refactor patterns using constant values of reference types, such as Strings and Dates, as enumeration members. Lastly, we will compliment our plug-in with

[11] *inter-program* or *inter-module* in the case of COBOL.

usage statistics collection so that we may gain more insight into how the tool is used in practice and if and where any improvements can be made.

Acknowledgments

We would like to thank Jason Sawin, Atanas Rountev, and Frank Tip for answers to our technical questions and for referring us to related work.

References

Bäumer D, Gamma E, Kiezun A (2001) Integrating refactoring support into a Java development tool. In: OOPSLA'01 Companion

Bloch J (2001) Effective Java Programming Language Guide. Prentice Hall PTR

Bracha G, Cohen N, Kemper C, Odersky M, Stoutamire D, Thorup K, Wadler P (2003) Adding generics to the Java programming language: Public draft specification, version 2.0. Tech. Rep. JSR 014, Java Community Process

Dean J, Grove D, Chambers C (1995) Optimization of object-oriented programs using static class hierarchy analysis. In: European Conference on Object-Oriented Programming, pp 77–101

Deursen AV, Moonen L (1998) Type inference for COBOL systems. In: Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, pp 220–230, DOI 10.1109/WCRE.1998.723192

Deursen AV, Moonen L (1999) Understanding cobol systems using inferred types. In: International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, pp 74–81, DOI 10.1109/WPC.1999.777746

Dig D, Marrero J, Ernst MD (2009) Refactoring sequential java code for concurrency via concurrent libraries. In: International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, ICSE '09, pp 397–407, DOI 10.1109/ICSE.2009.5070539

von Dincklage D, Diwan A (2004) Converting Java classes to use generics. In: Object-Oriented Programming, Systems, Languages & Applications, pp 1–14

Donovan A, Kiezun A, Tschantz MS, Ernst MD (2004) Converting Java programs to use generic libraries. In: Object-Oriented Programming, Systems, Languages & Applications, pp 15–34

Eidorff PH, Henglein F, Mossin C, Niss H, Sørensen MH, Tofte M (1999) Annodomini: From type theory to year 2000 conversion tool. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, USA, pp 1–14, DOI http://doi.acm.org/10.1145/292540.292543

Fowler M (1999) Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional

Fuhrer R, Tip F, Kiezun A, Dolby J, Keller M (2005) Efficiently refactoring Java applications to use generic libraries. In: European Conference on Object-Oriented Programming, pp 71–96

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston, MA, USA

Gosling J, Joy B, Steele G, Bracha G (2005) Java^{TM}Language Specification (3rd Edition). Addison-Wesley

Gravley JM, Lakhotia A (1996) Identifying enumeration types modeled with symbolic constants. In: Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, p 227

Kerievsky J (2004) Refactoring to Patterns. Pearson Higher Education

Khatchadourian R, Muskalla B (2010) Enumeration refactoring: A tool for automatically converting java constants to enumerated types. In: Int. Conf. Automated Software Engineering, ACM, New York, NY, USA, ASE '10, pp 181–182, DOI 10.1145/1858996.1859036

Khatchadourian R, Sawin J, Rountev A (2007) Automated refactoring of legacy Java software to enumerated types. In: Int. Conf. Software Maintenance, IEEE, ICSM 2007, pp 224–233, DOI 10.1109/ICSM.2007.4362635

Kiezun A, Ernst MD, Tip F, Fuhrer RM (2007) Refactoring for parameterizing Java classes. In: International Conference on Software Engineering

Knuth DE (1967) Semantics of context-free languages. Theory of Computing Systems 2(2)

Kumar A, Sutton A, Stroustrup B (2012) Rejuvenating C++ programs through demacrofication. In: Int. Conf. Software Maintenance, IEEE, pp 98–107

Oracle Corporation (2010) Java programming language: Enhancements in jdk 5. URL `http://docs.oracle.com/javase/1.5.0/docs/guide/language`

Oracle Corporation (2015) Typesafe enums. URL `http://docs.oracle.com/javase/8/docs/technotes/guides/language/enums.html`

Palsberg J, Schwartzbach MI (1994) Object-oriented type systems. John Wiley and Sons Ltd., Chichester, UK

Pierce BC (2002) Types and programming languages. MIT Press

Ramalingam G, Field J, Tip F (1999) Aggregate structure identification and its application to program analysis. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, New York, NY, USA, pp 119–132, DOI http://doi.acm.org/10.1145/292540.292553

Steimann F, Siberski W, K'uhne T (2003) Towards the systematic use of interfaces in java programming. In: PPPJ

Steimann F, Mayer P, Meißner A (2006) Decoupling classes with inferred interfaces. In: ACM Symposium on Applied Computing, pp 1404–1408

Sutter BD, Tip F, Dolby J (2004) Customization of java library classes using type constraints and profile information. In: European Conference on Object-Oriented Programming, pp 585–610

Tip F, Kiezun A, Bäumer D (2003) Refactoring for generalization using type constraints. In: Object-Oriented Programming, Systems, Languages & Applications, pp 13–26

Tip F, Fuhrer R, Dolby J, Kiezun A (2004) Refactoring techniques for migrating applications to generic Java container classes. Tech. Rep. RC 23238, IBM T.J. Watson Research Center

Vallée-Rai R, et al (2000) Optimizing Java bytecode using the Soot framework: Is it feasible? In: International Conference on Compiler Construction, LNCS 1781, pp 18–34