Really Automatic Scalable Object-Oriented Reengineering

Marco Trudel[1], Carlo A. Furia[1], Martin Nordio[1], and Bertrand Meyer[1,2]

[1] Chair of Software Engineering, ETH Zurich, Switzerland
[2] Software Engineering Laboratory, ITMO, St. Petersburg, Russia
Email: `firstname.lastname@inf.ethz.ch`

Abstract. Even when implemented in a purely procedural programming language, properly designed programs possess elements of good design that are expressible through object-oriented constructs and concepts. For example, placing structured types and the procedures operating on them together in the same module achieves a weak form of encapsulation that reduces inter-module coupling. This paper presents a novel technique, and a supporting tool AutoOO, that extracts such implicit design elements from C applications and uses them to build reengineered object-oriented programs. The technique is completely automatic: users only provide a source C program, and the tool produces an object-oriented application written in Eiffel with the same input/output behavior as the source. An extensive evaluation on 10 open-source programs (including the editor `vim` and the math library `libgsl`) demonstrates that our technique works on applications of significant size and builds reengineered programs exhibiting elements of good object-oriented design, such as low coupling and high cohesion of classes, and proper encapsulation. The reengineered programs also leverage advanced features such as inheritance, contracts, and exceptions to achieve a better usability and a clearer design. The tool AutoOO is freely available for download.

1 Introduction

The reasons behind the widespread adoption of object-oriented programming languages have to be found in the powerful mechanisms they provide, which help design and implement clear, robust, flexible, and maintainable programs. Classes, for example, are modular constructs that support strong encapsulation, which makes for components with high cohesion and low coupling; inheritance and polymorphism make classes extensible, thus promoting flexible reuse of implementations; exceptions can handle inter-procedural behavior without polluting functional and modular decomposition; and contracts seamlessly integrate specification and code, and support abstract yet expressive designs.

Competent programmers, however, try to achieve the same design goals—encapsulation, extensibility, and so on—even when they are implementing in a programming language that does not offer object-oriented features. A developer adopting the C programming language, for example, will use files as primitive modules collecting **struct**s and functions operating on them; will implement exception handling through a disciplined use of *setjmp* and *longjmp*; will use conditional checks and defensive programming to define valid calling contexts in a way somewhat similar to preconditions.

Following these observations, this paper describes work to automatically reengineer procedural C programs to introduce object-oriented features, based on design elements such as files, function signatures, and user-defined types. The result of our work is a fully automated technique and supporting tool that extract such *implicit* design information from C programs[3] and use it to *reengineer* functionally equivalent *object-oriented* applications in the Eiffel object-oriented programming language.

Given the huge availability of high-quality C applications, an automatic technique to reengineer C into object-oriented code has a major potential practical impact: reusing legacy code in modern environments. In fact, this is not the first attempt at supporting object-oriented reengineering, and porting procedural applications to a modern programming paradigm is a recurrent industrial practice. A careful analysis of related work, which we present in Section 8, shows however that previous approaches have limitations in terms of comprehensiveness, automation, applicability to real code, and achieved quality of the reengineering. In contrast, our approach constitutes a significant contribution with the following distinguishing characteristics.

- The reengineering technique is *fully automatic* and implemented in the freely available tool AutoOO. Users only need to provide an input C project; AutoOO outputs an object-oriented Eiffel application that can be compiled.
- The technique and tool work on real software of considerable size, as demonstrated by an extensive evaluation on 10 open-source programs including the editor `vim` and the math library `libgsl`.
- As demonstrated by quantitative analysis of the products of the automatic reengineering, the object-oriented code achieves good encapsulation and introduces inheritance, contracts, and exceptions when feasible.
- The reengineering is correct by construction: the generated object-oriented programs achieve the same functional behavior as the source programs and do not introduce potentially incorrect refactorings that might break the code.

These characteristics make AutoOO a valuable asset to reuse good-quality software in object-oriented environments. We have experienced the usefulness of this service first-hand with the Eiffel user community, which is not as large as those of other mainstream languages, and hence it lacks a wide choice of libraries in some application domains. Prompted by numerous requests, in addition to the 10 programs discussed in Section 5, we used AutoOO to port some C libraries that were sorely needed by the Eiffel developer's community: the driver for the MongoDB database; the PCRE regular expression library; and the SDL mixer audio library. After being produced with minimal effort, the Eiffel versions of these libraries are now being used by Eiffel developers. Requests for converting more libraries keep coming, and AutoOO is starting to be directly used by programmers other than its authors. This gives us confidence that our work is practical and helps solve a real and recurrent problem: automatic and scalable reuse.

While AutoOO translates C to Eiffel, the principles and reengineering techniques it implements are based on standard object-oriented features, and hence are readily applicable to other programming languages—such as C++, Java, and C#—offering classes,

[3] We target ANSI C and GCC extensions.

static members, visibility modifiers, and exceptions.[4] In fact, to highlight the generality of the reengineering, the presentation will use a Java-like syntax; this will be palatable to readers familiar only with C-based programming languages without misrepresenting any conceptually relevant aspect. We assume knowledge of the standard terminology and notions of object-oriented programming [15].

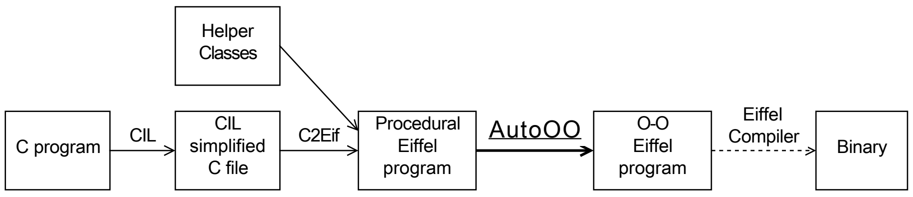

Fig. 1. Object-oriented reengineering with C2Eif and AutoOO.

C2Eif and AutoOO. The reengineering techniques described in this paper are combined with our previous work on C2Eif [27] and implemented as part of the toolchain shown in Figure 1. The toolchain implements an overall transformation that inputs a C program and outputs an object-oriented Eiffel project with the same functionality. The C source program is first processed by CIL [19], which simplifies some C constructs (for example, there is only one type of loop in CIL). In the second stage, C2Eif transliterates the CIL output into procedural Eiffel, whose structure replicates that of the C input program without introducing elements of object-oriented design. We described this stage in previous work [27]; its details are largely independent of the reengineering techniques implemented by AutoOO, which only uses C2Eif as a back-end. Finally, AutoOO processes the C2Eif output, introduces the transformations described in Sections 3 and 4, and outputs the reengineered object-oriented Eiffel programs that can be compiled.

Tool availability. AutoOO is available online at `http://se.inf.ethz.ch/research/c2eif`. The webpage includes AutoOO's sources, pre-compiled binaries, source and binaries of all translated programs of Table 2, and a user guide. *AutoOO's distribution has been successfully evaluated by the ECOOP artifact evaluation committee and found to meet expectations.* For ease of presentation, we will use the name AutoOO to denote not only the tool but also the reengineering technique it implements.[5]

[4] The only two features used by AutoOO that may not be universally available are contracts and member renaming during inheritance. Contracts, however, are increasingly provided in other languages as libraries or assertions (e.g., CodeContracts for C#, **assert** in Java). Member renaming plays a limited role in the refactorings produced by AutoOO (Section 3.4), and a translator targeting another language could make up for it by using the same name in the super- and subclasses, or even (as we suggest in Section 3.4) by dropping inheritance in the few cases where renaming is required.

[5] In the latest distributions, C2Eif and AutoOO are integrated into a single translator (called C2Eif for simplicity), which offers the option to apply the object-oriented reengineering transformations presented in this paper.

Outline. In the rest of the paper, Section 2 defines the goals of AutoOO reengineering, how they are assessed, and the design principles followed. Section 3 discusses how AutoOO introduces elements of object-oriented design—in particular, how it populates classes. Section 4 discusses how it introduces contracts and exceptions. Section 5 presents the evaluation of the correctness, scalability, and performance of AutoOO based on 10 reengineered applications and libraries. Section 6 reviews the fundamental aspects of the object-oriented design style introduced by AutoOO and how they make for usable reengineered programs. Section 7 discusses the current limitations of AutoOO. Section 8 reviews related work and compares AutoOO against existing tools and approaches to object-oriented reengineering. Section 9 concludes and outlines future work.

2 O-O Reengineering: Goals, Principles, and Evaluation

The overall goal of AutoOO reengineering is *expressing* the design *implicit* in procedural programs using constructs and properties of the object-oriented paradigm. For example, we restructure and encapsulate the code into classes that achieve a high cohesion and low coupling, we make use of inheritance to reuse code, and so on. The main motivation for introducing object-oriented constructs is that they can *explicitly* express design and structure of programs concisely and in a way amenable to further flexible extension and reuse.

While reengineering in its most general meaning—the reconstruction of "a system in a new form" [2]—may also introduce new functionality or mutate the existing one (for example, with corrective maintenance), the present work tries not to deviate from the original intentions of developers as reflected in the procedural implementations.[6] For example, we do not introduce exceptions unless the original program defines some form of inter-procedural execution path. We adopt a conservative approach because we want a reengineering technique that:

- is *completely automatic*, not just a collection of good practices and engineering guidelines;
- always produces *correct* reengineerings, that is programs that are functionally equivalent to the original procedural programs.

Improving and extending software are important tasks, but largely orthogonal to our specific goals and requiring disparate techniques. For example, there are serviceable tools to infer specifications from code (to mention just a few: [5,12,28]) which can be applied atop our reengineering technique to get better code specification automatically; but including them in our work would weaken the main focus of the contribution.

From the user perspective, AutoOO is a translator that takes an input C program and converts it to an object-oriented Eiffel program that replicates its functionality. The rest of this section presents other specific goals of AutoOO and how we assess them.

[6] This entails that, when applied to C programs that do not contain elements conducive to object-oriented design, AutoOO should simply introduce few changes. The experiments with the programs of Table 1 suggest, however, that AutoOO's heuristics are often applicable with success.

Case studies. The evaluation of AutoOO, described in the following sections, targets 10 open-source programs totalling 750 KLOC. The 10 programs include 7 applications and 3 libraries; all of them are widely-used in Linux and other "*nix" distributions. `hello world` is the only toy application, which is however useful as a baseline. The other applications are: `micro httpd` 12dec2005, a minimal HTTP server; `xeyes` 1.0.1, a widget for the X Windows System that shows two googly eyes following the cursor movements; `less` 382-1, a text terminal pager; `wget` 1.12, a command-line utility to retrieve content from the web; `links` 1.00, a simple web browser; `vim` 7.3, a powerful text editor. The libraries are: `libcurl` 7.21.2, a URL-based transfer library supporting protocols such as FTP and HTTP; `libgmp` 5.0.1, for arbitrary-precision arithmetic; `libgsl` 1.14, a powerful numerical library. Section 5 discusses more details about the programs used in the experiments.

Correctness, scalability, and performance. In addition to systematic interactive usage, we assess *correctness* of the reengineering produced by AutoOO by running the standard regression test-suites available with the programs, hereby verifying that the output is the same in C and Eiffel. We also consider the *translation time* taken by AutoOO to guarantee that it scales up; and the *performance* of the Eiffel reengineered program to ensure that it does not incur a slowdown that severely compromises usability. Section 5 discusses these correctness and performance results.

Object-oriented design. AutoOO creates an object-oriented program consisting of a collection of classes; each class aggregates data definitions (fields) and functions operating on them (methods). Section 3 presents the technique that extracts object-oriented design; we evaluate the quality of the object-oriented design produced by AutoOO with the following measures:

Soundness: we manually inspected 43% of all classes produced by AutoOO (all projects but `vim` and `libgsl`) and we determined how many methods belong to the correct class, that is are indeed methods operating on the fields of the class.[7]

Coupling and cohesion: the coupling of a class is measured as the ratio: number of accesses to members of other classes / number of accesses to members of the same class. When this ratio is low (less than 1 in the best cases), it shows that classes are loosely coupled and with high cohesion.[8]

Information hiding is measured as the ratio of private to public members. A high ratio indicates that classes make good usage of information hiding for encapsulation.

Instance vs. class members: the ratio of instance to class members (called **static** members in Java) gives an idea of the "object-orientedness" of a design. A high ratio indicates a really *object* oriented design, as it makes limited usage of "global" class fields and methods.

Inheritance: we manually inspected all uses of inheritance introduced by AutoOO and we determined how many correctly define substitutable heir classes.

[7] As we illustrate in Section 3, soundness refers to whether reengineering moves members to the "right" classes from a design point of view. Soundness is thus a notion orthogonal to *correctness*: AutoOO reengineerings do not alter behavior and hence are always correct (as per standard regression testsuites and general usage).

[8] Cohesion is normally defined as the dual of coupling.

Contracts and exceptions. In addition to the core elements of object-oriented design, AutoOO also introduces high-level features often present in object-oriented languages: contracts and exceptions. AutoOO clearly distinguishes the purpose of contracts vs. exceptions.

Contracts replace annotations (not part of ANSI C but available as GCC extensions) and encode simple requirements on a function's input and guarantees on its output; they are discussed in Section 4.1.

Exceptions replicate the behavior of *setjmp* and *longjmp* which divert the structured control flow in exceptional cases across functions and modules; they are discussed in Section 4.2.

3 Object-Oriented Design

REENGINEERING STEP	# bundle methods	# datatype methods	# bundle fields	# datatype fields	% sound datatype methods	average coupling	overall hiding	instance/class members	# inheriting classes
1. source files	12,445	0 (0%)	3,628	5,337 (60%)	–	8.87 1.54	–	0.33	0
2. function signature	7,724	4,721 (38%)	3,628	5,337 (60%)	94%	2.33 1.20	–	0.88	0
3. call graph	6,471	5,974 (47%)	2,881	6,084 (68%)	96%	2.00 1.06	0.12	1.12	0
4. inheritance	6,471	5,974 (47%)	2,881	6,084 (68%)	96%	2.00 1.06	0.12	1.12	4

Table 1. Object-oriented design metrics after each reengineering step applied to the ten case study programs.

AutoOO produces object-oriented designs that consist of collections of classes. The generated classes are of two kinds with different purposes:

- a *datatype class* combines the data definitions translating some C type definition (**struct** or **union**) with a collection of instance methods translating C functions operating on the type.
- a *bundle class* collects global variables and global functions present in some C source file and makes them available to clients as class members.

Only datatype classes are germane to object-oriented design, which emphasizes proper encapsulation of data definitions with the operations defined on them; bundle classes, however, are still necessary to collect elements that do not clearly belong exclusively to any datatype, such as globals shared by multiple clients. Thus, bundle classes are a safe fall-back that keeps the original modular units (the source files) instead of forcing potentially unsound refactorings.

Corresponding to their roles, datatype classes contain mainly "proper" *instance* members (Section 3.3 discusses the exceptions), whereas bundle classes contain only *class* members (also called **static**).

AutoOO generates datatype and bundle classes in four steps:

1. *Source file* analysis creates the bundle classes and populates them based on the content of source files; it creates a datatype class for each structured type definition (**struct** or **union**).
2. *Function signature* analysis refactors methods from bundle to datatype classes, moving operations closer to the data definition they work on.
3. *Call graph* analysis refactors members from bundle to datatype classes, and shuffles methods among datatype classes, moving members to classes where they are exclusively used.
4. *Inheritance* analysis creates inheritance relationships between datatype classes based on their fields.

The following subsections 3.1–3.4 describe the steps in detail with examples.

Table 1 reports how the various metrics mentioned in Section 2 change as we apply the four steps to the 10 case study programs. For each reengineering step, Table 1 reports:

– The number of bundle and datatype members[9] created, partitioned in methods and fields.
– The percentage of *sound* datatype methods.[10] A method *m* of a datatype class *T*—that contains the data definition of a **struct** *T* or **union** *T*—is *sound* if manual analysis confirms that *m* implements an operation whose primary purpose is modifying or querying instances of *T*.
– The average (median) *coupling* of classes, where the coupling of a class *T* (with respect to the rest of the system) is defined as follows. An *access* is the read or write of a field, or a method call; an access in the body of a method *m* of *T* is *in* if it refers to a member of *T* other than *m*; it is *out* if it refers to a member of a class other than *T*. When counting accesses in a method *m* we ignore duplicates: if *m*'s body calls *r* more than once, we only count it as one access. *T*'s coupling is the ratio of out to in accesses of all its members. For each step, Table 1 reports two values of coupling; the value on top puts all classes of all programs together (hence larger projects dominate), while the bottom value computes medians per programs and then the median across programs.
– The *hiding* of classes, measured as the ratio of **private** and **protected** to **public** members.
– The ratio of *instance* to *class* members.
– The number of classes defined using *inheritance*.

The rest of this section discusses the figures shown in Table 1 to demonstrate how each reengineering step improves these object-oriented design metrics. A word of caution is necessary about the reliability of metrics such as those we use to assess the

[9] A bundle (or datatype) member is a member of a bundle (or datatype) class.
[10] Evaluated on all projects but `vim` and `libgsl`, as discussed in Section 2.

improvement of design quality, something which eludes general quantitative definitions [3]. Nonetheless, metrics give an idea of how the design changes through the various transformations; while the exact values they report should be taken with a grain of salt, they are still useful to complete the picture of how AutoOO performs in practice.

3.1 Source File Analysis

For each source file `F.c` in the program, the first reengineering step creates a bundle class *F* and populates it with translations of all the global variables and function definitions found in `F.c`. For each definition of a structured type *T* in `F.c`, the first step also creates a datatype class *T* that contains *T*'s components as fields. AutoOO only has to consider structured type definitions using **struct** or **union**; atomic type definitions and **enum**s are handled in the initial processing by C2Eif. Since AutoOO's reengineering treats the two kinds of structured type declarations uniformly, we only deal with **struct**s in the following to streamline the presentation; the handling of **union**s follows easily.

```
int majority_age = 18;

struct person
{
    int age;
    bool sex;
};

void set_age(struct person *p, int new_age) {
    if(new_age ≤ 0) return;
    p→age = new_age;
}

bool overage(int age) {
    return (age > majority_age);
}

bool is_adult(struct person *p) {
    return overage(p→age);
}
```

Fig. 2. C source file `PersonHandler.c`.

For example, when processing the C source file in Figure 2, the first step generates the datatype class *Person* and the bundle class *PersonHandler* in Figure 3.

Source file analysis sets up the dual bundle/datatype design and defines the classes of the system. The result is still far from good object-oriented design as the datatype classes are just empty containers mapping **struct**s one-to-one, and in fact we have

```
class Person
{
    int age;

    boolean sex;
}
```

```
class PersonHandler
{
    static int majority_age = 18;

    static void set_age(Person p, int new_age) {
        if(new_age≤0) return;
        p.age = new_age;
    }

    static boolean overage(int age) {
        return (age > majority_age);
    }

    static boolean is_adult(Person p) {
        return overage(p.age);
    }
}
```

Fig. 3. Datatype class *Person* (left) and bundle class *PersonHandler* (right) initially created for the C file in Figure 2.

no hiding and a low instance/class member ratio (the only instance members are the datatype fields).

The overall coupling (first row in Table 1) is also quite high after step 1. This does not come as a surprise: because all methods are located in bundle classes, every read or write of a **struct** field from the original C code becomes an out access. The proliferation of out accesses is especially evident in `libgmp`, where the majority of modules have *only* out accesses. In general, coupling is higher for libraries in our experiments; this may indicate that coupling for library code should be measured differently, for example by considering the library in connection with a client. In any case, this is not a problem for our evaluation: the value of coupling after step 1 is merely a baseline that corresponds to purely procedural design; our goal is to measure how this value changes as we apply the next reengineering steps.

3.2 Function Signature Analysis

The second reengineering step moves methods from bundle to datatype classes according to their signature, with the intent of having data and methods operating on them in the same class.

Consider a method *m* of bundle class *M* with signature

$$t_0 \; m \; (t_1 \; p_1, t_2 \; p_2, \ldots, t_n \; p_n)\,,$$

for $n \geq 0$. An argument p_k of *m* is *data-bound* if its type $t_k = T*$ (pointer to *T*), where *T* is a datatype class. When a routine has more than one such argument, we consider only the first one in signature order. A data-bound argument p_k is *globally used* by *m* if it is accessed (read or written) at least once along every path of *m*'s control flow graph, except possibly for argument handling paths. An *argument handling path* is a path guarded by a condition that involves some argument p_h, with $h \neq k$, and terminated by a **return**.

For each method *m* of a bundle class *M* that has a data-bound argument p_k of type $t_k = T*$ which is globally used, the second reengineering step moves *m* into the datatype class *T* and changes its signature—which becomes non-**static** and drops argument p_k—and its body—which refers to p_k implicitly as **this**. Accordingly, *m*'s body may have to adjust other references to members of *M* that are now in a different class; also any call to *m* has to be adjusted following its new signature.

Continuing the example of Figure 2, the second reengineering step determines that *set_age* and *is_adult* can be refactored: argument *p* is data-bound and globally used in both methods (with the first instruction in *set_age* being an argument handling path). Hence, the two methods are moved from class *PersonHandler* to class *Person* which becomes:

```
class Person
{
    int age;

    boolean sex;

    void set_age(int new_age) {
        if(new_age≤0) return;
        age = new_age;
    }

    boolean is_adult() {
        return PersonHandler.overage(age);
    }
}
```

Function signature analysis introduces fundamental elements of object-oriented design. As reported in Table 1, manual inspection reveals that 94% of the methods moved to datatype classes are indeed operations on that type; this means that 97% of the members of datatype classes (fields plus sound methods) are refactored correctly. Remember that our definition of soundness refers to design, not to correct behavior: even the $6\% = 100\% - 94\%$ "unsound" methods behave correctly as in the original C programs, even if they are arguably not allocated to the best class. Inspection also reveals some common causes of unsound refactorings. Some functions use a generic pointer (type **void**$*$) as first argument, and then cast it to a specific **struct**$*$ in the code; and in a few cases the pointer arguments are simply not reliable indicators of data dependence or are in the wrong order (more details below).

Coupling drastically reduces after step 2, because many methods that access fields of datatype classes are now located inside those classes. This dominates over the increase in out accesses to bundle members from within the methods moved to datatype classes, also introduced by step 2. In particular, function signature analysis mitigates the high coupling we measured in the libraries. Finally, many methods have become instance methods, with an overall instance/class ratio of 0.88.

How restrictive is the choice to consider only the first data-bound argument to a datatype class for deciding where to move methods? For example, if the code in Figure 2 had another function **void** *do_birthday*(**struct** *person* $*p$, **struct** *log* $*l$) that increases p's age and writes to the log pointed to by l, should we move *do_birthday* to datatype class *log* instead of *person*? The empirical evidence we collected suggests that our heuristics is generally not restrictive: we manually analyzed all 77 functions with multiple arguments of type "pointer to **struct**" in the case study programs and found only 3 cases where the "sound" refactoring would target an argument other than the first.

Another feature of function signature analysis as it is implemented in AutoOO is the choice to ignore methods with an argument p whose type t corresponds to a **struct**, but that is passed by *copy* (in other words, whose original type in C is t rather than $t*$); we found 131 such cases among the programs of Table 2 and only 56 (43%) of them would have generated a sound refactoring. In all, we preferred not to consider arguments passed by copy because it would lead to unsound refactoring in the majority of cases; a more sophisticated analysis of this aspect belongs to future work.

Finally, the refactoring requirement that a data-bound argument must be globally used is not necessary, in most cases, to achieve soundness, but dropping it would introduce incorrect translations that change the behavior of the program in some cases. In fact, a function with an argument not used globally includes valid executions where the argument is allowed to be **null**; therefore, it cannot become an instance method which always has an implicit non-**null** target **this**.

As an interesting observation about the application of reengineering to the programs of Table 2, we found that 40% of the methods moved to datatype classes in step 2 have a name that includes the datatype class name as prefix. For example, the methods operating on a datatype class *hash_table* in `wget` are named *hash_table_get*, *hash_table_put*, and so on. This suggests that, in the best cases, even purely syntactic information carries significant design choices. AutoOO takes advantage of this finding and removes such prefixes to increase the readability of the created code (see also the client example in Section 6).

3.3 Call Graph Analysis

The third reengineering step moves more members to datatype classes according to where the members are used, with the intent of encapsulating "utility" members together with the datatype definitions that use them exclusively.

Consider a member n of any class N that is accessed (read, written, or called) *only* in a datatype class T. For each such member n, the third reengineering step moves n into the datatype class T. If n is an instance method or a class method, it becomes an instance method; if it is a class field, it remains a class field to preserve the original semantics of **static** fields corresponding to global C variables (this is the only case where we add class members to datatype classes). Members moved to datatype classes in this step also become **private**, since they are not used outside the class they are moved to. Since moving a member out of a class changes the global call graph, AutoOO performs the third reengineering step iteratively: it starts with the member *n* with the largest number of accesses, and updates the call graph after every refactoring move, recalculating the set of candidate members for the next move.

Continuing the example of Figure 2, assume that method *overage* of bundle class *PersonHandler* is only called by *is_adult* in datatype class *Person*, and that field *majority_age* is instead read also by other modules. Then, AutoOO moves *overage* to *Person* where it becomes non-**static** and **private**:

```
private boolean overage(int age) {
    return (age > PersonHandler.majority_age);
}
```

The field *majority_age*, instead, stays unchanged in class *PersonHandler*.

As reported in Table 1, call graph analysis refines the object-oriented design and introduces hiding when possible, that is for 12% of the members. Even if there are 2,290 private members, these are localized in only 139 classes, hence the average hiding per class is low (3% mean). Coupling decreases once more, as a result of moving utility methods to the class where they are used. The percentage of sound refactored methods increases to 96%; overall, 98% of the datatype members are refactored correctly. Step 3 also makes instance members the majority (53% of all members, or 1.12 instance member per class member).

Under the conservative approach taken by AutoOO, which creates functionally equivalent code, the values of hiding, coupling, and instance/class members reached after steps 1–3 strike a fairly good balance between introducing object-oriented features and preserving the original design as not to harm understandability due to unsound members in classes of the reengineered application. The example in Section 6 reinforces these conclusions from a user's perspective.

3.4 Inheritance Analysis

The fourth reengineering step introduces inheritance in order to make existing subtyping relationships between datatype classes explicit. In the original C code subtyping surfaces in the form of casts between different **struct** pointer types. Because the language does not provide any way to make one **struct** type conform to another, modelling subtyping in C requires frequent *upcasting* (conversion from a subtype to a supertype) as well as *downcasting* (from a supertype to a subtype). Inheritance analysis finds such casting patterns and establishes inheritance relationships between the involved types.

Consider two type declarations in the source C program:

struct *r* { $t_1\ a_1; t_2\ a_2; \ldots ; t_m\ a_m;$ };
struct *s* { $u_1\ b_1; u_2\ b_2; \ldots ; u_n\ b_n;$ };

We say that type *s* is *cast* to type *r* if there exists, anywhere in the program's code, a cast of the form (**struct** *r*∗) *e* with *e* an expression of type **struct** $s*$. We say that type s *extends* type *r* if $n > m$[11] and, for all $1 \leq i \leq m$, the types t_i and u_i are equivalent.

[11] The case $n = m$ could be also supported but would rarely be useful with the programs tried so far.

For every such types *r* and *s* such that *s* extends *r* and *s* is cast to *r*, *r* is cast to *s*, or both, the fourth reengineering step makes the datatype class for *s* inherit from *r*. Using a **renames** clause[12] to rename fields with different names, *s* becomes:

```
class s extends r renames $a_1$:$b_1$, $a_2$:$b_2$, ..., $a_m$:$b_n$
{
    $u_{m+1}$ $b_{m+1}$;
    ...
    $u_n$ $b_n$;
    // Rest of the class unchanged.
}
```

Notice that AutoOO bases inheritance analysis on type information only, not on field *names*. Therefore, it requires renaming of fields in general; implementing this feature in Java or similar languages, where renaming is not possible, would require some workaround (or simply dropping inheritance when renaming is required).

Continuing the example of Figure 2, assume another **struct** declaration is **struct** *student* { **int** *age*; **bool** *sex*; **int** *gpa*; } and that, somewhere in the program, a variable of type *person* $*$ is cast to (*student* $*$). Then, datatype class *Student* becomes:

```
class Student extends Person
{
    int gpa;
    /*...*/
}
```

While AutoOO identified 1,875 pairs t_1, t_2 of types where t_1 extends t_2, and 96 pairs where t_1 is cast to t_2, only 4 pairs satisfy both requirements. Hence, the introduction of inheritance in our experiments is limited to 4 classes (2 in each of `xeyes` and `less`). This is largely a consequence of the original C design where extensions of **struct**s along these lines are infrequent, combined with the constraint that our reengineering create functionally equivalent code and be automatic. All few uses of inheritance AutoOO identified are, however, sound, in that the resulting types are real subtypes that satisfy the substitution principle. In contrast, manual inspection reveals that none of the other $92 = 96 - 4$ pairs of cast types determine classes that are related by inheritance. Introducing inheritance for the other 1,871 pairs solely based on one type extending the other is most likely unsound without additional evidence. Many **struct**s, for example, are collections of integer fields, but they model semantically disparate notions that are not advisable to combine. The other metrics in Table 1 do not change after inheritance analysis, assuming we count fields in the *flattened* classes.

Interestingly, the two instances of inheritance we found in `less` use renaming to define lists as simplified header elements. For example:

[12] Available natively in Eiffel and not in Java, but whose semantics is straightforward.

```
struct element_list {                struct element {
    struct element *first;               struct element *next;
};                                       char *content;
                                     };
```

The two types are indeed compatible, and the renaming makes the code easier to understand even without comments.

4 Contracts and Exceptions

AutoOO introduces contracts and exceptions to improve the readability of the classes generated in the reengineering. Section 4.1 explains how AutoOO builds contracts from compiler-specific function annotations and from simple implicit properties of pointers found by static analysis. Section 4.2 discusses how exceptions can capture the semantics of *longjmp*.

4.1 Contracts

Contracts are simple formal specification elements embedded in the program code that use the same syntax as Boolean expressions and are checked at runtime. AutoOO constructs two common kinds of contracts that annotate methods, namely preconditions and postconditions. A method's precondition (introduced by **requires**) is a predicate that must hold whenever the method is called; it is the caller's responsibility to establish the method's precondition before calling it. A method's postcondition (introduced by **ensures**) is a predicate that must hold whenever the method terminates; it is the method's body responsibility to guarantee the postcondition upon termination.

AutoOO creates contracts from two information sources commonly available in C programs:

– GCC function attributes;
– globally used pointer arguments.

Based on these, AutoOO added 3,773 precondition clauses and 13 postcondition clauses to the programs in Table 2.

GCC function attributes. The GCC compiler supports special function annotations with the keyword ***__attribute__***. GCC can use these annotations during static analysis for code optimization and to produce warnings if the attributes are found to be violated. Among the many annotations supported—most of which are relevant only for code optimization, such as whether a function should be inlined—AutoOO constructs preconditions from the attribute ***nonnull*** and postconditions from the attribute ***noreturn***. The former specifies which of a function's arguments are required to be non-**null**; the latter marks functions that never return (for example, the system function *exit*). For each method *m* $(t_1\ p_1, \dots, t_m\ p_m)$ corresponding to a C function with attribute ***nonnull*** $(i_1, \dots, i_n)$, with $n \geq 0$ and $1 \leq i_1, \dots, i_n \leq m$ denoting arguments of *m* by position, AutoOO adds to *m* the precondition

requires $p_{i_1} \neq$ **null**, $p_{i_2} \neq$ **null**, ..., $p_{i_n} \neq$ **null**

that the arguments $p_{i_1}, \dots, p_{i_n}$ be non-**null**. For each method *m* corresponding to a C function with attribute ***noreturn***, AutoOO adds to *m* the postcondition **ensures false** that would be violated if *m* ever terminates.

Extending the example of Figure 2, the function:

```
__attribute__ (( nonnull (2), noreturn ))
void kill(struct person *p, struct person *q) {
    /*...*/
    printf("A person is killed at age %d", q→age);
    exit(1);
}
```

gets the following signature after reengineering (assuming the first argument becomes **this**):

void *kill*(*Person q*)   **requires** $q \neq$ **null**   **ensures false**

GCC function attributes determined 266 precondition and 13 postcondition clauses in the programs of Table 2.

Globally used pointers. Section 3.2 defined the notion of *globally used* argument: an argument that is accessed (read or written) at least once along every path in a method's body. Based on the same notions, for each pointer argument *p* of a method *m* that is globally used in *m* on *all* paths (including argument-handling paths), AutoOO adds to *m* the precondition **requires** $p \neq$ **null** that *p* be non-**null**. The precondition does not change the behavior of the method: if *m* were called with *p* = **null**, *m* would eventually crash in every execution when accessing a **null** reference, and hence $p \neq$ **null** is a necessary condition for *m* to correctly execute.

Through globally used pointer analysis, AutoOO introduced 3,507 precondition clauses in the programs of Table 2.

Defensive programming is a programming style that tries to detect violations of implicit preconditions and takes countermeasures to continue execution without crashes. For example, when function *set_age* in Figure 2 is called with a non-positive *new_age*, it returns without changing *p*'s age field, thus avoiding corrupting it with an invalid value. While defensive programming and programming with contracts have similar objectives—defining necessary conditions for correct execution—they achieve them in very different ways: while contracts clearly specify the semantics of interfaces and assign responsibilities for correct execution, defensive programming just tries to communicate failures while working around them. This fundamental difference is the reason why we do not use contracts to replace instances of defensive programming when reengineering: doing so would change the behavior of programs. In the case of *set_age*, for example, a precondition **requires** $new\_age > 0$ would cause the program to terminate with an error whenever the precondition is violated, whereas the C implementation continues execution without effects. In addition, C functions often use integer return arguments as error codes to report the outcome of a procedure call; introducing contracts would make clients incapable of accessing those codes in case of error.

Relaxed contracts for memory allocation. The GCC distribution we used in the experiments provides *__attribute__* annotations (see Section 4.1) also for system libraries. In particular, the memory allocation functions *memcpy* and *memmove*:

```
__attribute__ (( nonnull (1, 2) ))
extern void *memcpy(void *dest, const void *src, size_t n);

__attribute__ (( nonnull (1, 2) ))
extern void *memmove(void *dest, const void *src, size_t n);
```

require that their pointer arguments *dest* and *src* be non-**null**. By running the reengineering produced by AutoOO, we found that this requirement is often spuriously violated at runtime: when the functions are called with the third argument *n* equal to 0, they return without accessing either *dest* or *src*, which can therefore safely be **null**. Correspondingly, AutoOO builds the contracts for these functions a bit differently:

requires $n == 0 \,||\, (dest \neq$ **null** && $src \neq$ **null**)

that is *dest* and *src* must be non-**null** only if *n* is non-zero. This inconsistency in GCC's annotations does not have direct effects at runtime in C because annotations are not checked. We ignore whether it might have other subtle undesirable consequences as the compiler may use the incorrect information to optimize binaries.

4.2 Exceptions

Object-oriented programming languages normally include dedicated mechanisms for handling exceptional situations that may occur during execution. While error handling is possible also in procedural languages such as C, where it is typically implemented with functions returning special error codes, exceptions in object-oriented languages are more powerful because they can traverse the call stack searching for a suitable handler; this makes it possible to easily cross the method and class boundaries in exceptional situations, without need to introduce a complex design that harms the natural modular decomposition effective in all non-exceptional situations.

C programmers can explicitly implement a similar mechanism that jumps across function boundaries with the library functions *setjmp* (save an arbitrary return point) and *longjmp* (jump back to it). AutoOO detects usages of these library functions and renders them using exceptions in the object-oriented reengineering. AutoOO defines a helper class *CE_EXCEPTION* which can use Eiffel's exception propagation mechanism to go back in the call stack to the allocation frame of the method that called *setjmp*. There, local jump instructions reach the specific point saved with *setjmp* within the method's body. We do not discuss the details of the translation because they refer to several low-level mechanisms discussed in [27] that are out of scope in the present paper. From the point of view of the reengineering, however, the translation expresses the complex semantics of *longjmp* naturally through the familiar exception handling mechanism.

AutoOO found 6 usages of *longjmp* in the programs of Table 2, which it replaced with exceptions.

We did not make a more extensive usage of exceptions, for example for replacing return error codes. In many cases, it would have complicated the object-oriented design and slowed down the program, without significant benefits. A fine-grained analysis of the instances of defensive programming, with the goal of selecting viable candidates that can be usefully translated through exceptions, belongs to future work.

5 Correctness, Scalability, and Performance

In addition to the metrics of object-oriented design displayed in Table 1 and discussed in the previous sections, we evaluated the *behavior* of the reengineering produced by AutoOO on the 10 programs in Table 2. All the experiments ran on a GNU/Linux box (kernel 2.6.37) with a 2.66 GHz Intel dual-core CPU and 8 GB of RAM, GCC 4.5.1, CIL 1.3.7, EiffelStudio 7.0.8.

	SIZE (LOCS)			TRANS-	BINARY
	PROCEDURAL	O-O	#	LATION	SIZE
	(C)	(EIFFEL)	CLASSES	(S)	(MB)
`hello world`	8	15	1	1	1.1
`micro httpd`	565	1,983	16	1	1.3
`xeyes`	1,463	10,665	77	1	1.6
`less`	16,955	22,709	75	5	2.3
`wget`	46,528	61,040	178	24	4.1
`links`	70,980	108,726	227	31	12.5
`vim`	276,635	414,988	669	138	22.6
`libcurl`	37,836	70,413	272	17	–
`libgmp`	61,442	82,379	223	20	–
`libgsl`	238,080	378,025	729	81	–
TOTAL	750,492	1,150,943	2,467	319	45.5

Table 2. Reengineering of 10 open-source programs.

The reengineering of each program proceeds as previously shown in Figure 1, with the end-to-end process (from C source to object-oriented Eiffel output) being push-button.

For each program used in our evaluation, Table 2 reports: the size of the source procedural program in C (after processing by CIL); the size of the reengineered object-oriented program in Eiffel output by AutoOO; the number of classes generated by the reengineering; the source-to-source time taken by the reengineering (including both C2Eif's translation and AutoOO's reengineering, but excluding compilation of Eiffel output to binary); the size of the binary after compiling the Eiffel output with EiffelStudio[13].

Correctness. In all cases, the output of AutoOO successfully compiles with EiffelStudio without need for any adjustment or modification. After compilation, we ran extensive trials on the compiled reengineered programs to verify that they behave as in

[13] In EiffelStudio, libraries cannot be compiled without a client.

their original C version. We performed some standard usage sessions with the interactive applications (`xeyes`, `less`, `links`, and `vim`) and verified that they behave as expected and they are usable interactively. We also performed systematic usability tests for the other applications (`hello world`, `micro httpd`, and `wget`) which can be used for batch processing; and ran standard regression testsuites (also automatically translated from C to Eiffel) on the libraries. All usability and regression tests execute and pass on both the C and the translated Eiffel versions of the programs, with the same logged output.

Scalability of the reengineering process is demonstrated by the moderate translation times (second to last column in Table 2) taken by AutoOO: overall, reengineering 750 KLOC of C code into 1.1 MLOC of Eiffel code took less than six minutes.

Performance. We compared the performance of AutoOO's reengineered output against C2Eif's non-reengineered output for the non-interactive applications and libraries of Table 2. The performance is nearly identical in C2Eif and AutoOO for all programs but the `libgsl` testsuite, which even executed 1.33 times *faster* in the reengineered AutoOO version. This shows that the object-oriented reengineering produced by AutoOO improves the design without overhead with respect to a bare non-reengineered translation. The basic performance overhead of switching from C to Eiffel—analyzed in detail in [27]—significantly varies with the program type but, even when it is pronounced, it does not preclude the usability of the translated application or library in standard conditions. These conclusions carry over to programs reengineered with AutoOO, and every optimization introduced in the basic translation provided by C2Eif will automatically result, at the end of the tool chain, in faster reengineered applications.

6 Discussion: AutoOO's Object-Oriented Style

All object-oriented designs produced by AutoOO deploy a collection of classes partitioned into bundle and datatype classes, as explained in Section 3. While this prevents a more varied gamut of designs from emerging as a result of the automatic reengineering, in our experience it does not seem to hamper the readability and usability of the reengineered programs, as we now briefly demonstrate with a real-world example. We attribute this largely to the fact that AutoOO produces *sound* reengineering in most cases, and programs with correct behavior in all cases. Therefore, the straightforward output design is understandable by programmers familiar with the application domain, who can naturally extend or modify it to introduce new functionality or a more refined design.

As mentioned in Section 1, we have distributed to the Eiffel developers community a number of widely used C libraries translated with AutoOO. One of them is MongoDB, a document-oriented (non-relational) database[14]. Consider a client application that uses MongoDB's API to open a connection with a database and retrieve and print all documents in a collection *tutorial.people*. Following the API tutorial, this could be written in C as shown in Figure 4 on the left. A client using the MongoDB library translated

[14] `http://www.mongodb.org/display/DOCS/C+Language+Center`

and reengineered by AutoOO would instead use the syntax shown in Figure 4 on the right.

```
 1  // connect to database                   // connect to database
 2  mongo conn[1];                           Mongo conn = new Mongo();
 3  int status = mongo_connect(conn,         int status = conn.connect(
 4               "127.0.0.1", 27017);                     "127.0.0.1", 27017);
 5
 6  // iterate over database content         // iterate over database content
 7  mongo_cursor cursor[1];                  MongoCursor cursor = new MongoCursor();
 8  mongo_cursor_init(cursor, conn,          cursor.init(conn.address,
 9               "tutorial.people");                      "tutorial.people");
10  while(mongo_cursor_next(cursor)          while(cursor.next() == MONGO_OK)
11         == MONGO_OK) {                    {
12     bson_print(&cursor→current);             cursor.current.print();
13  }                                        }
14  mongo_cursor_destroy(&cursor);           Mongo.mongo_cursor_destroy(cursor.address);
15
16  // disconnect from database              // disconnect from database
17  mongo_destroy(conn);                     conn.destroy();
```

Fig. 4. A MongoDB client application written in C (left) and the same application written for the AutoOO translation of MongoDB (right).

On the one hand, the two programs in Figure 4 are structurally similar, which entails that users familiar with the C version of MongoDB will have no problem switching to its object-oriented counterpart, and would still be able to understand the C documentation in the new context. On the other hand, the program on the right nicely conforms to the object-oriented idiom: variable definitions are replaced by object creations (lines 2 and 7); and function calls become instance method calls (lines 3, 8, 12, and 17). Method names are even more succinct, because they lose the prefixes "*mongo_*" and "*mongo_cursor_*" unnecessary in the object-oriented version where the type of the target object conveys the same information more clearly.

The only departure from traditional object-oriented style is the call to the cursor destruction function on line 14, which remains a **static** method call with identical signature. AutoOO did not turn it into an instance method because its implementation can be called on **null** pointers, in which case it returns without any effect:

```
int mongo_cursor_destroy(mongo_cursor *cursor) { if(!cursor) return 0; /* ... */ }
```

As discussed in Section 3.2, AutoOO does not reengineer such functions because the target of an object-oriented call is not allowed to be **null**. In such cases, users may still decide that it is safe to refactor by hand such examples; in any case, the AutoOO translation provides a proper reengineering of most of the library functionalities.

7 Limitations

By and large, the evaluation with the programs of Table 2 demonstrates that AutoOO is a scalable technique applicable to programs of considerable size and producing good-quality object-oriented designs automatically. This section discusses the few limitations that remain, distinguishing between those of the underlying C to Eiffel translation and those of the object-oriented reengineering.

C to Eiffel translation. As discussed in detail in [27], the raw translation from C to Eiffel provided by C2Eif does not currently support: a few rare programming patterns that rely on specific memory layouts, such as how the arguments passed to a function are stored next to one another; and a few GCC exotic extensions. Using CIL as preprocessor, while it contributes to simplifying and maintaining the translation, also carries its own limitations: K&R legacy C is not supported; and comments are stripped and formatting is lost, and hence this information cannot be used to improve the readability and formatting of the translated Eiffel code. None of these limitations is intrinsic to the AutoOO approach, and lifting them is largely an engineering effort: we plan to remove the dependency on CIL as well as to support additional non-standard features of the C language if they will be often needed by users of AutoOO.

Object-oriented reengineering. All the limitations of our reengineering technique follow the decision to be conservative, that is not to change the behavior in any case, to only extract design information already present in the C programs, and to only introduce refactorings with empirically demonstrated high success rates, in terms of accurately capturing design elements. For example, Section 3.2 discussed how a refactoring based on struct arguments passed by copy would lead to less than 50% of sound refactorings, while we normally aim at success rates over 90%.

While these requirements make it possible to have a robust and fully automatic technique, they may also be limiting in some specific cases where users are willing to push the reengineering, accepting the risk of having to revise the output of AutoOO before using it.

Formal correctness proofs. A final limitations of our work is the lack of formal correctness proofs of the basic C translation and of the reengineering steps. While the evaluation (discussed in Section 5) extensively tested the translated applications without finding any unexpected behavior—which gives us good confidence in the robustness of the results—this still falls short of a fully formal approach such as [4]. This is planned as future work.

8 Related Work

Reengineering [2] is a common practice—and an expensive activity [22]—in professional software development. Given the wide adoption of languages with object-oriented features, object-oriented reengineering is frequently necessary. In this section, we briefly review some general literature on reengineering of legacy systems (8.1), followed by a detailed analysis of significant approaches to object-oriented reengineering (8.2). For lack of space, we do not include a general review of *refactoring* techniques and methods [7], as the focus of this paper is extracting object-oriented designs automatically from the analysis of procedural code rather than refactoring per se.

	target language	completely automatic	available	readability	external libraries	pointer arithmetic	gotos	inlined assembly
`Ephedra` [14]	Java	no	no	+	no	no	no	no
`C2J++` [26]	Java	no	no	+	no	no	no	no
`C2J` [21]	Java	no	yes	−	no	yes	no	no
`C++2Java` [25]	Java	no	yes	+	no	no	no	no
`C++2C#` [25]	C#	no	yes	+	no	no	no	no
AutoOO	Eiffel	yes	yes	+	yes	yes	yes	yes

Table 3. Tools translating C to object-oriented languages.

8.1 Reengineering of Legacy Systems

The main goal in reengineering a legacy system is raising the level of abstraction. Typically, this is achieved by *translating* an implementation written in an old programming language—such as K&R C, Fortran-77, or old COBOL—into a modern programming language such as Java [16,1,29]. This process does not normally include improving the object-oriented design but only making the same system available in a supported environment.

To summarize the state-of-the art in this area, Table 3 lists five tools that translate C to an object-oriented language *without object-oriented reengineering* and compares them against AutoOO. The table is taken from our previous work on C2Eif [27], the C to Eiffel translator on top of which we built AutoOO—which therefore appears as last entry of the table. For each tool, Table 3 reports (see [27] for more details):

– The *target language*.
– Whether the tool is *completely automatic*, that is whether it generates translations that are ready for compilation without need for any manual rewrite or adaptation.
– Whether the tool is *available* for download and usable.
– An assessment of the *readability* of the code produced.
– Whether the tool supports unrestricted calls to *external libraries*, unrestricted *pointer arithmetic*, unrestricted **goto**s, and inlined *assembly code*.

Even at the level of bare translation of C programs without object-oriented reengineering, the currently available tools do not support the full C language used in real programs because they cannot translate features such as external libraries and unrestricted pointer arithmetic, whose exact behavior is very complicated to get right but is necessary to have fully automatic translation tools. A recent comparative evaluation covering a wide range of tools for legacy system reengineering [18] points to similar limitations that prevent achieving complete automation. AutoOO, in contrast, can count on C2Eif's full support of the complete C language used in real programs, which underpins the development of a robust and scalable object-oriented reengineering tool.

8.2 Object-Oriented Reengineering

Among the broad literature on reengineering for modern systems, we identified nine approaches that target specifically object orientation. Table 4 summarizes their main features and compares them with ours. Following the primary goals of our work, described in Section 2, Table 4 lists:

- The *source* and the *target* languages (or if it is a generic methodology).
- Whether *tool support* was developed, that is whether there exists a tool or the paper explicitly mentions the implementation of a tool. A YES in small caps denotes the only currently publicly available tool, namely AutoOO.
- Whether the approach is *completely automatic*, that is if it performs O-O reengineering without any user input other than providing a source procedural program.
- Whether the approach supports the *full* source *language* or only a subset thereof.
- Whether the approach has been *evaluated*, that is whether the paper mentions evidence, such as a case study, that the approach was tried on real programs. If available, the table indicates the size of the programs used in the evaluation.
- Whether the approach performs *class identification*, that is if it groups fields and methods in classes.
- Whether the reengineering technique introduces object-oriented features, namely it identifies *instance methods* (as opposed to class methods which should have a restricted role in object orientation) and uses of *inheritance*.

Table 5 gives some notes about *limitations* of the approaches.

	source–target	tool support	completely automatic	full language	evaluated	class identification	instance methods	inheritance
Gall [9]	methodology	no	no	–	yes	yes	?	no
Jacobson [10]	methodology	no	no	–	yes	yes	no	–
Livadas [13]	C–C++	yes	no	no	no	yes	yes	no
Kontogiannis [11]	C–C++	yes	no	?	10KL	yes	?	yes
Frakes [8]	C–C++	yes	no	no	2KL	yes	?	no
Fanta [6]	C++–C++	yes	no	no	120KL	yes	?	no
Newcomb [20]	Cobol–OOSM	yes	yes	no	168KL	yes	?	no
Mossienko [17]	Cobol–Java	yes	no	no	25KL	yes	no	no
Sneed [23]	Cobol–Java	yes	yes	no	200KL	yes	?	no
Sneed [24]	PL/I–Java	yes	yes	no	10KL	yes	?	no
AutoOO	C–Eiffel	YES	yes	yes	750KL	yes	yes	yes

Table 4. Comparison of approaches to O-O reengineering.

[20] and [23] are the only authors that report evaluations on code bases of significant size. [20]'s reengineering, however, produces OOSM (hierarchical object-oriented state

	LIMITATIONS
Gall [9]	requires assistance of human expert
Jacobson [10]	only defines a process; 3 case studies from industry
Livadas [13]	prototype implementation; no support for pointers
Kontogiannis [11]	sound reengineering for only about 36% of the source code
Frakes [8]	translation may change the behavior; requires expert judgement
Fanta [6]	requires expert judgement
Newcomb [20]	only a model is generated, no program code
Mossienko [17]	only partial automation; the translation may change the behavior
Sneed [23]	domain-specific translation
Sneed [24]	domain-specific translation

Table 5. Overview of limitations.

machine models) models; mapping OOSM to a standard compilable object-oriented language is not covered. [23] reports that some manual corrections of the automatically generated Java code were necessary during the translation of the code base, although these manual interventions were later implemented as an extension of the translator; anyway, [23] targets the translation of domain-specific applications and in fact does not support the full input language; the authors of [23] expect that tackling new applications will require extending the tool.

[11]'s approach to introduce inheritance is based on the analysis of **struct** fields and of function signatures. AutoOO also uses **struct** field analysis to introduce inheritance, but limits the analysis to field types and ignores field names (see Section 3.4).

A direct detailed comparison of other tools with AutoOO on specific object-oriented features is difficult to obtain as several of these works focus on some aspects of the reengineering but provide few concrete details about other aspects or about how the reengineering is performed on real code. This is also the reason for the presence of "?" in the column "instance methods", corresponding to cases where we could not figure out the details of how methods are refactored. In light of the evidence collected (or lack thereof), it is fair to say that identifying instance methods automatically without expert judgement is an open challenge; and so are full source language support and complete automation. These features are a novel contribution of AutoOO.

9 Conclusions and Future Work

We presented a new completely automatic approach to object-oriented reengineering of C programs and a freely available supporting tool AutoOO. AutoOO scales to applications and libraries of significant size, and produces reengineered object-oriented programs that are directly compilable and usable with the same behavior as the source C programs. The reengineered object-oriented designs produced by AutoOO encapsulate fields and methods operating on them with a high degree of soundness—thus lowering coupling and increasing cohesion—and make judicious usage of inheritance, contracts, and exceptions to improve the quality of the object-oriented design.

Future work. The remaining limitations of AutoOO, discussed in Section 7, suggest items for future work:

- When we recognize the usage of standard library services (e.g., generic data structures), we will replace them with their Eiffel counterparts when possible, extending existing approaches for mapping APIs [30].
- Section 3 discussed possible additional sources of information to improve the amount of sound reengineered methods in datatype classes (e.g., **struct** arguments passed by copy). The empirical data we collected in our experiments suggest, however, that these sources would frequently lead to unsound refactorings if directly applied. For these cases, we will investigate more sophisticated analyses that try to understand in which cases a sound refactoring is possible.
- Finally, we will explore the possibility of combining AutoOO's completely automatic approach with additional user input (e.g., domain knowledge useful to understand the software design), with the goal of tailoring the reenginering to each application.

Acknowledgments. This work was partially supported by the ETH grant "Object-oriented reengineering environment".

References

1. B. L. Achee and D. L. Carver. Creating object-oriented designs from legacy FORTRAN code. *JSS*, 39(2):179–194, 1997.
2. E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
3. M. Ó. Cinnéide, L. Tratt, M. Harman, S. Counsell, and I. H. Moghadam. Experimental assessment of software metrics using automated refactoring. In *ESEM*, pages 49–58. ACM, 2012.
4. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.
5. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.
6. R. Fanta and V. Rajlich. Reengineering object-oriented code. In *International Conference on Software Maintenance, 1998. Proceedings*, pages 238 –246, 1998.
7. M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999.
8. W. Frakes, G. Kulczycki, and N. Moodliar. An empirical comparison of methods for reengineering procedural software systems to object-oriented systems. In *ICSR*, volume 5030 of *LNCS*, pages 376–389, 2008.
9. H. Gall and R. Klosch. Finding objects in procedural programs: an alternative approach. In *WCRE*, pages 208–216. IEEE, 1995.
10. I. Jacobson and F. Lindström. Reengineering of old systems to an object-oriented architecture. In *OOPSLA*, pages 340–350. ACM, 1991.
11. K. Kontogiannis and P. Patil. Evidence driven object identification in procedural code. In *STEP*, pages 12–21. IEEE, 1999.
12. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.

13. P. E. Livadas and T. Johnson. A new approach to finding objects in programs. *Journal of Software Maintenance*, 6(5):249–260, 1994.
14. J. Martin and H. A. Müller. Strategies for migration from C to Java. In *CSMR*, pages 200–210. IEEE Computer Society, 2001.
15. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
16. R. Millham. An investigation: reengineering sequential procedure-driven software into object-oriented event-driven software through UML diagrams. In *COMPSAC*, pages 731–733, 2002.
17. M. Mossienko. Automated Cobol to Java recycling. In *CSMR*, pages 40–50. IEEE, 2003.
18. B. S. Nadera, D. Chitraprasad, and V. S. S. Chandra. The varying faces of a program transformation systems. *ACM Inroads*, 3(1):49–55, Mar. 2012.
19. G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, pages 213–228, 2002.
20. P. Newcomb and G. Kotik. Reengineering procedural into object-oriented systems. In *WCRE*, pages 237–249. IEEE, 1995.
21. Novosoft. C2J: a C to Java translator. `http://www.novosoft-us.com/solutions/product_c2j.shtml`, 2001.
22. H. Sneed. Planning the reengineering of legacy systems. *Software, IEEE*, 12(1):24 –34, jan 1995.
23. H. Sneed. Migrating from COBOL to Java. In *ICSM*, pages 1–7. IEEE, 2010.
24. H. Sneed. Migrating PL/I code to Java. In *CSMR*, pages 287–296. IEEE, 2011.
25. Tangible Software Solutions. C++ to C# and C++ to Java. `http://www.tangiblesoftwaresolutions.com/`.
26. E. Tilevich. Translating C++ to Java. In *German Java Developers' Conference Journal*. Sun Microsystems Press, 1997.
27. M. Trudel, C. A. Furia, M. Nordio, B. Meyer, and M. Oriol. C to O-O translation: Beyond the easy stuff. In *Proceedings of WCRE*, pages 19–28. IEEE, 2012.
28. Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, pages 191–200. ACM, 2011.
29. A. Yeh, D. Harris, and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *WCRE*, pages 227–236, 1995.
30. H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *ICSE*, pages 195–204, 2010.