Diss. ETH No. 21166

Automatic Translation and Object-Oriented Reengineering of Legacy Code

A dissertation submitted to
ETH ZÜRICH

for the degree of
Doctor of Sciences

presented by
MARCO TRUDEL
Master of Science ETH in Informatik, ETH Zürich, Switzerland
Ingenieur FH in Informationstechnologie, ZHW, Switzerland

born
July 18th, 1983

citizen of
Männedorf, ZH

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Hausi A. Müller, co-examiner
Prof. Dr. Richard Paige, co-examiner
Dr. Manuel Oriol, co-examiner

2013

Acknowledgments

First and foremost I would like to thank Bertrand Meyer for giving me the opportunity to do a PhD under his supervision at ETH Zurich. His support and the research freedom he granted me made the work presented here possible.

I also thank the co-examiners of this dissertation, Hausi Müller, Richard Paige and Manuel Oriol, who kindly made time in their busy schedules to co-referee this work.

The other present and past members of the Chair of Software Engineering have been more a family than mere co-workers for me in this journey, so my thanks also go to them: Andrey Rusakov, Benjamin Morandi, Carlo A. Furia, Christian Estler, Claudia Günthart, Cristiano Calcagno, Jason (Yi) Wei, Julian Tschannen, Manuel Oriol, Marco Piccioni, Martin Nordio, Max (Yu) Pei, Mei Tang, Michela Pedroni, Mischael Schill, Scott West, Sebastian Nanz, Stephan van Staden, and last but not least Nadia Polikarpova for sharing more than just the PhD journey with me.

It was a great ride! Thanks to everyone who was involved in some way.

Financing

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

Contents

Abstract

Can we reuse some of the huge code-base developed in C to take advantage of modern programming language features such as type safety, object-orientation, and contracts? This dissertation presents a source-to-source translation and object-oriented reengineering of C code into Eiffel, a modern object-oriented programming language, and the supporting tool C2Eif. The migration is completely automatic and supports the entire C language (ANSI, as well as many GNU C Compiler extensions) as used in practice, including pointer arithmetic, usage of native system libraries and inlined assembly code. Eiffel programs are created exhibiting elements of good object-oriented design, such as low coupling and high cohesion of classes, and proper encapsulation. The programs also leverage advanced features such as inheritance, contracts, and exceptions to achieve a better usability and a clearer design. Our experiments show that C2Eif can handle C applications and libraries of significant size (such as `vim` and `libgsl`), as well as challenging benchmarks such as the GCC torture tests. The produced Eiffel code is functionally equivalent to the original C code, and takes advantage of some of Eiffel's features to produce safe and easy-to-debug programs.

We have also investigated the related problem of automatically translating source code between different modern environments, in particular from Java to Eiffel. The translation has been formalized in order to increase confidence in its correctness, and the usability of the supporting tool J2Eif has been evaluated on four programs of varying complexity.

Zusammenfassung

Ist es möglich einen Teil der riesigen in C entwickelten Code-Basis wiederzuverwenden um die Vorteile von modernen Programmiersprachen wie Typsicherheit, Objektorientierung und Verträge nutzen zu können? Diese Dissertation präsentiert eine Quelltext-zu-Quelltext Übersetzung und objektorientierte Umstrukturierung von C Code nach Eiffel, eine moderne objektorientierte Programmiersprache, und das unterstützende Tool C2Eif. Die Migration ist völlig automatisch und unterstüzt die gesamte C Sprache (ANSI sowie viele GNU C Compiler Erweiterungen) wie sie in der Praxis verwendet wird, einschliesslich Zeigerarithmetik, der Benutzung von nativen System-Bibliotheken und Inline-Assembler-Code. Eiffel Programme werden erstellt die Elemente von gutem objektorientiertem Design besitzen wie lose Kopplung und starke Bindung von Klassen und geeignete Kapselung. Die Programme setzen auch fortschrittliche Merkmale ein wie Vererbung, Verträge und Ausnahmen um eine bessere Benutzerfreundlichkeit und ein klareres Design zu erreichen. Unsere Experimente zeigen dass C2Eif C Anwendungen und Bibliotheken von signifikanter Grösse (wie z.B. `vim` und `libgsl`) sowie anspruchsvolle Benchmarks wie die GCC Folter Tests bearbeiten kann. Der erzeugte Eiffel Quelltext ist funktional äquivalent zu dem ursprünglichen C Quelltext und nutzt einige Eigenschaften von Eiffel um sichere und einfach zu korrigierende Programme zu erstellen.

Wir haben ebenfalls das ähnliche Problem einer automatischen Quelltext-zu-Quelltext Übersetzung zwischen verschiedenen modernen Sprachen untersucht, namentlich von Java zu Eiffel. Die Übersetzung wurde formalisiert um das Vertrauen in die Korrektheit zu erhöhen und die Benutzbarkeit des unterstützenden Tools J2Eif wurde an vier Programmen mit unterschiedlicher Komplexität getestet.

Chapter 1

Overview and Main Results

1.1 Motivation

Programming languages have significantly evolved since the original design of C in the 1970's as a "system implementation language" [40] for the Unix operating system. C was a high-level language by the standards of the time, but it is decidedly low-level compared to modern programming paradigms, as it lacks advanced features—static type safety, encapsulation, inheritance, and contracts [28], to mention just a few—that impact programmers productivity as well as software quality and maintainability.

Given the huge availability of high-quality C programs, an automatic technique to translate C into object-oriented code has a major potential practical impact: reusing legacy code in modern environments.

1.2 Overview

This dissertation is part of the research effort of improving existing approaches for legacy code migration, proposing new ones and evaluating them. The basis for this work is C2Eif: a fully automatic migration framework for C. It implements a source-to-source translation and object-oriented reengineering into the Eiffel programming language.

Whenever possible, C2Eif translates C language constructs into equivalent Eiffel constructs; this is the case for functions (called *routines* in Eiffel), variables, statements, expressions, loops, and basic types with arithmetic operations. In cases where direct counterparts are not available, the tool offers simulations using existing constructs and library support. For instance, jump statements (**break**, **continue**, **return** and **goto**) are simulated

using structured control flow and auxiliary variables. Library support is offered for pointers: they get translated into instances of a generic library class *CE_ POINTER* [*G*], which supports the full C pointer functionality. C structs are translated into Eiffel classes, which inherit from a library class *CE_ CLASS*. Using reflection, this class provides means to convert its instance into an object with the exact memory layout prescribed by the C struct, which is necessary for pointer arithmetic and interoperability with precompiled C libraries.

Object-oriented reengineering extracts elements of good design present in high-quality C code and expresses them through object-oriented constructs and concepts. Reengineering in C2Eif consists of four steps exploiting such implicit design elements: (1) *source file* analysis creates classes and populates them based on the decomposition of code into source files, (2) *function signature* analysis reassigns routines to classes they work on, (3) *call graph* analysis reassigns class members (called *features* in Eiffel) to classes where they are exclusively used, (4) *inheritance* analysis creates inheritance relationships among classes based on their attributes. In addition to these core elements of object-oriented design, contracts and exceptions —high-level features often present in object-oriented languages— are also introduced, based on GNU C Compiler (GCC) annotations, requirements on the input of functions, and usages of the *setjmp* library.

Our work is not the first attempt to support object-oriented translation. Porting procedural programs to a modern programming paradigm is a recurring industrial practice. A survey of related work, which we present in Chapter 2, shows, however, that previous approaches have limitations in terms of comprehensiveness, automation, applicability to real code, and quality of the translation. In contrast, our approach constitutes a significant contribution with the following distinguishing characteristics.

- The translation is *fully automatic* and implemented in a freely available tool, C2Eif. Users only need to provide an input C project; C2Eif outputs an object-oriented Eiffel program that can be compiled and executed.

- C2Eif does not stop at the core features but extends over the often difficult "last mile": it covers the entire C language as used in practice. Examples include pointer arithmetic, calls to pre-compiled C libraries (e.g., for I/O), inlined assembly, and unrestricted branch instructions including **goto** and *setjmp*/*longjmp*.

- The technique and the tool have been applied to real software of considerable size. An extensive evaluation on 13 open-source programs (in-

cluding the editor `vim` and the math library `libgsl`) demonstrates that our translation produces object-oriented code with high level of encapsulation and introduces inheritance, contracts, and exceptions where appropriate.

- The translation is correct by construction: it does not use potentially unsound refactorings, and thus the generated programs exhibit the same functional behavior as the source programs.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 contains an overview of the state of the art in legacy code migration, discussing the work that is most closely related to the contributions of this dissertation. Chapter 3 describes the way C2Eif bridges the language barrier and creates functionally correct translations. Chapter 4 is devoted to AutoOO: a part of C2Eif, which reengineers programs to achieve high-quality object-oriented designs. Chapter 5 concentrates on translations between modern programming languages and presents J2Eif: a fully automatic source-to-source translator of Java into Eiffel. Finally, Chapter 6 draws conclusions and lists directions for future research.

Chapter 2

State of the Art

There are two main approaches to reusing source code written in a "foreign" language inside a different "host" language: creating wrappers for the components written in the foreign language; and translating foreign source code into functionally equivalent host code. Table 2.1 provides an overview of the advantages and disadvantages of the two approaches in terms of:

- Code quality: does the resulting code exhibit proper design and style; is it easy to understand and maintain?
- Correctness: is the translated program guaranteed to be functionally equivalent to the source program?
- Effort: is the approach cost-effective?

The next sections discuss the comparison in more detail and present additional issues of reusing source code of different languages.

	Wrapping		Translation	
	manual	automatic	manual	automatic
Code Quality	−	−	+	?
Correctness	+	+	−	?
Effort	−	+	−−	+

Table 2.1: Comparison of code reuse approaches; +/−/? stands for positive/negative/tool-dependent.

2.1 Wrapping foreign code

Wrappers enable reuse of foreign implementations through the API of bridge libraries. This approach (e.g., [8, 7, 41]) does not modify the foreign code, whose functionality is therefore not altered; moreover, the complete foreign language is supported. On the other hand, types of data that can be retrieved through the bridging API are often restricted to a simple subset common to the host and the foreign language (e.g., primitive types).

We judge the resulting code quality of the wrapping approach negatively: using several languages within the same software system hinders understandability and requires programmers to be fluent in all of them. Maintenance is also problematic because debugging is complicated by the use of multiple runtime environments.

Correctness, on the other hand, is generally not a problem since the foreign code is reused without modifications. There is a risk of introducing errors in the interface code (if the wrapping is performed manually), but the amount and complexity of this code is usually insignificant compared to the wrapped code, and consequently, the errors are less severe.

The approach is relatively low-effort, except that manual wrapping can be quite tedious.

2.2 Translating foreign code

Industrial practices have long encompassed manual migration of legacy code. Several semi-automated tools exist that help translate code written in legacy programming languages such as old versions of COBOL [49, 29], PL/IX [19], Fortran-77 [1, 46], and K&R C [53]. Further work in this area proposes using domain-specific knowledge [14] as well as testing and visualization techniques [39] to help develop the translations.

Some translators focus on the adaptation of code into an extension (superset) of the original language. Examples include the migration of legacy code to object-oriented code, such as Cobol to OO-Cobol [35, 43, 52], Ada to Ada95 [47], and C to C++ [20, 54]. Some of these efforts go beyond the mere hosting of the original code, and introduce refactorings that take advantage of the object-oriented paradigm. Most of these refactorings are, however, limited to restructuring modules into classes.

Ephedra [25, 23] is a tool that translates legacy C to Java. It first translates K&R C to ANSI C; then, it maps data types and type casts; finally, it translate the C source code to Java. Ephedra handles a significant subset of C, but it does not translate frequently used features such as unrestricted

gotos, external pre-compiled libraries, and inlined assembly code. A case study evaluating Ephedra [26] involved three small programs: the implementation of *fprintf*, a monopoly game (about 3 KLOC), and two graph layout algorithms. The study reports that the source programs had to be manually adapted to be processable by Ephedra.

Other tools, proprietary or open-source, that translate C (and C++) to Java or C# include: Convert2Java [2], C2J++ [50], C2J [37], and C++2Java and C++2C# [48]. Table 2.2 contains a comparison of all currently available translators, showing:

- The *target language*.
- Whether the tool is *completely automatic*, that is, whether it generates translations that are ready for compilation.
- Whether the tool is *available* for download and usable; in some cases we could only find a paper describing the tool, but were not able to obtain an implementation that would work on a standard machine.
- An assessment (subjective to a certain extent) of the *readability* of the resulting code. In particular, we took into account if the translated code is sufficiently similar to the C source, to be readily understandable by a programmer familiar with the latter. We judged `C2J` negatively on this point because the tool stores all program data in a single global array in order to support pointer arithmetic. Not only is this detrimental to readability, but also circumvents type checking in the Java translation.
- Whether the tool supports arbitrary calls to *external libraries*, full *pointer arithmetic*, unrestricted **goto**s, and inlined *assembly code*.

	target language	completely automatic	available	readability	external libraries	pointer arithmetic	**goto**s	inlined assembly
`Ephedra`	Java	no	no	+	no	no	no	no
`Convert2Java`	Java	no	no	+	no	no	no	no
`C2J++`	Java	no	no	+	no	no	no	no
`C2J`	Java	no	yes	−	no	yes	no	no
`C++2Java`	Java	no	yes	+	no	no	no	no
`C++2C#`	C#	no	yes	+	no	no	no	no

Table 2.2: Tools translating C to O-O languages.

None of the existing tools support all features of C listed in the last four columns of Table 2.2. Accurate translation of these features is hard to get

right; however, it is necessary for fully automatic migration of real-world C programs. A recent comparative evaluation covering a wide range of tools for legacy system reengineering [31] points to similar limitations that prevent achieving complete automation.

With manual translation, the resulting code quality is the biggest advantage. Assuming a careful reimplementation with a proper design as intended by the target language, the resulting code will have the best possible readability and will be easy to maintain. This comes at the expense of all other points though: the process is labour-intensive and time-consuming; it provides no correctness guarantees, and effectively forfeits years of testing and fixing errors in the name of code quality.

Using an automatic translation tool, on the other hand, is inexpensive and fast, but the achieved correctness and code quality depend on the tool. As demonstrated in Table 2.2, comprehensive and fully automatic translation tools are still beyond the state of the art.

2.3 Object-Oriented Reengineering

Reusing source code from a foreign language often implies a paradigm shift. In recent years, the most widespread example of such a shift has been reengineering of procedural code into object-oriented code.

Among the broad literature on reengineering for modern systems, we identified ten approaches that target specifically object orientation. Table 2.3 summarizes their main features, listing:

- The *source* and the *target* languages (not applicable to generic methodologies).
- Whether *tool support* is available.
- Whether the approach is *completely automatic*, that is, it does not require any user input other than the source procedural program.
- Whether the approach supports the *full* source *language* or only a subset thereof.
- Whether the approach has been empirically *evaluated* 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 attributes and routines into classes.

	source–target	tool support	completely automatic	full language	evaluated	class identification	instance routines	inheritance
Gall [14]	methodology	no	no	–	yes	yes	?	no
Jacobson [18]	methodology	no	no	–	yes	yes	no	–
Livadas [22]	C–C++	yes	no	no	no	yes	yes	no
Kontogiannis [20]	C–C++	yes	no	?	10KL	yes	?	yes
Frakes [13]	C–C++	yes	no	no	2KL	yes	?	no
Fanta [11]	C++–C++	yes	no	no	120KL	yes	?	no
Newcomb [35]	Cobol–OOSM	yes	yes	no	168KL	yes	?	no
Mossienko [29]	Cobol–Java	yes	no	no	25KL	yes	no	no
Sneed [45]	Cobol–Java	yes	yes	no	200KL	yes	?	no
Sneed [42]	PL/I–Java	yes	yes	no	10KL	yes	?	no

Table 2.3: Comparison of approaches to O-O reengineering.

	LIMITATIONS
Gall [14]	requires assistance of human expert
Jacobson [18]	only defines a process; three case studies from industry
Livadas [22]	prototype implementation; no support for pointers
Kontogiannis [20]	sound reengineering for only about 36% of the source code
Frakes [13]	translation may change the behavior; requires expert judgement
Fanta [11]	requires expert judgement
Newcomb [35]	only a model is generated, no program code
Mossienko [29]	only partial automation; translation may change the behavior
Sneed [45]	domain-specific translation
Sneed [42]	domain-specific translation

Table 2.4: Overview of limitations.

- Whether the reengineering technique introduces object-oriented features, such as *instance routines* (as opposed to class routines, which should have a restricted role in object orientation) and *inheritance*.

Table 2.4 summarizes the *limitations* of the approaches.

Only [35] and [45] report evaluation on code bases of significant size. The reengineering in [35], however, produces OOSM (hierarchical object-oriented state machine) models; mapping OOSM to an executable object-oriented language is not covered. [45] 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. In any case, [45] only targets applications from a specific domain and does not support the full input language; the authors of [45] expect that tackling new applications will require extending the tool.

As evidenced by Table 2.3, no comprehensive and fully automatic solution is currently available in the area of object-oriented reengineering.

2.4 Summary and Conclusions

In this chapter, we presented the two main approaches for reusing source code written in a foreign language within a different host language, namely wrapping and translation. Moreover, we discussed object-oriented reengineering, which is often necessary given the wide adoption of object-oriented features in modern programming languages and the procedural nature of legacy languages. We summarized the state of research in these areas and listed the advantages and disadvantages of the two code reuse approaches.

In conclusion, automatic translation of legacy code, including object-oriented reengineering where applicable, has clear advantages over the other approaches — provided tools are available that create correct, high-quality code. The present work is an attempt to advance the state of the art by developing such a tool; the results of the effort are presented in the following chapters.

Chapter 3

C to Eiffel Translation

3.1 Introduction

This chapter describes a fully automatic translation of C programs into Eiffel, an object-oriented programming language, together with a tool C2Eif, which implements the translation. While the most common approaches to re-use C code in other host languages are based on "foreign function APIs", source-to-source translation solves a different problem, and has some distinctive benefits: the translated code can take full advantage of the high-level nature of the target language and of its safer runtime.

Main features of C2Eif. Translating C to a high-level object-oriented language is challenging because it requires adapting to a more abstract memory representation, a tighter type system, and a sophisticated runtime that is not directly accessible. There have been previous attempts to translate C into an object-oriented language (see the survey in Section 2.2). A limitation of the resulting tools is that they hardly handle the trickier or specialized parts of the C language [9], which are tempting to dismiss as unimportant "corner cases", but figure prominently in real-world programs; examples include calls to pre-compiled C libraries (e.g., for I/O), inlined assembly, and unrestricted branch instructions including *setjmp* and *longjmp*.

One of the distinctive features of the present work is that it does not stop at the core features but extends over the often difficult "last mile": it covers the entire C language as used in practice. The completeness of the translation scheme is attested by the set of example programs to which the translation was successfully applied, as described in Section 3.4, including major applications such as the `vim` editor (276 KLOC), major libraries such as `libgsl` (238 KLOC), and the challenging "torture" tests for the GCC C compiler.

C2Eif is available at `http://se.inf.ethz.ch/research/c2eif`. The webpage includes C2Eif's sources, pre-compiled binaries, source and binaries of all translated programs of Table 3.1, and a user guide.

Sections 3.2–3.3 describe the distinctive features of the translation: it supports the complete C language (including pointer arithmetic, unrestricted branch instructions, and function pointers) with its native system libraries; it complies with ANSI C as well as many GNU C Compiler extensions through the CIL framework [32]; it is fully automatic, and it handles complete applications and libraries of significant size; the generated Eiffel code is functionally equivalent to the original C code (as demonstrated by running thorough test suites), and takes advantage of some advanced features, such as classes and contracts, to facilitate debugging of programming mistakes.

In our experiments, C2Eif translated completely automatically over 900,000 lines of C code from real-world applications, libraries, and testsuites, producing functionally equivalent[1] *Eiffel code.*

Safer code. Translating C code to Eiffel with C2Eif is quite useful to reuse C programs in a modern environment, but it also implies several valuable side-benefits—demonstrated in Section 3.4. First, the translated code blends well with hand-written Eiffel code because it is not a mere transliteration from C; it is thus modifiable with Eiffel's native tools and environments (EiffelStudio and related analysis and verification tools). Second, the translation automatically introduces simple contracts, which help detect recurring mistakes such as out-of-bound array access or null-pointer dereferencing. To demonstrate this, Section 3.4.4 discusses how we easily discovered a few unknown bugs in widely used C programs (such as `libgmp`) just by translating them into Eiffel and running standard tests. While the purpose of C2Eif is not to debug C programs, the source of errors is usually more evident when executing programs translated in Eiffel—either because a contract violation occurs, or because the Eiffel program fails sooner, before the effects of the error propagate to unrelated portions of the code. The translated C code also benefits from the tighter Eiffel runtime, so that certain buffer overflow errors are harder to exploit than in native C environments. Thus, Eiffel code generated by C2Eif is often safer and easy to maintain and debug.

3.2 Overview and Architecture

C2Eif is a compiler with graphical user interface that translates C programs to Eiffel programs. The translation is a complete Eiffel program that repli-

[1] As per standard regression testsuites and general usage.

cates the functionality of the C source program. C2Eif is implemented in Eiffel.

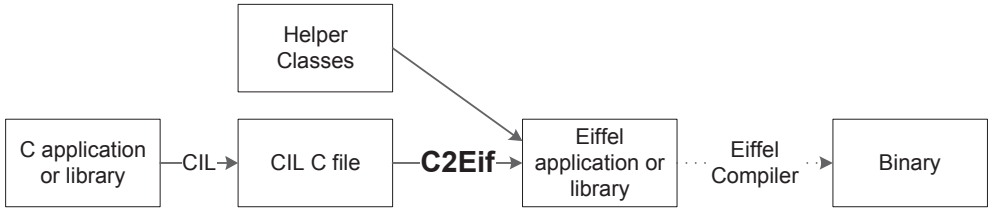

Figure 3.1: Translation with C2Eif.

High-level view. Figure 3.1 shows the overall picture of how C2Eif works. C2Eif inputs C projects (applications or libraries) processed with the C Intermediate Language (CIL) framework. CIL [32] is a C front-end that simplifies programs written in ANSI C or using the GNU C Compiler extensions into a restricted subset of C amenable to program transformations; for example, there is only one form of loop in CIL. Using CIL input to C2Eif ensures complete support of the whole set of C statements, without having to deal with each of them explicitly. C2Eif then translates CIL programs to Eiffel projects consisting of collections of classes that rely on a small set of Eiffel helper classes (described below). Such projects can be compiled with any standard Eiffel compiler.

Incremental translation. C2Eif implements a translation $\mathcal{T}$ from CIL C to Eiffel as a series $T_1, \ldots, T_n$ of successive incremental transformations on the Abstract Syntax Tree. Every transformation T_i targets exactly one language aspect (for example, loops or inlined assembly code) and produces a program in an intermediate language L_i which is a mixture of C and Eiffel constructs: the code progressively *morphs* from C to Eiffel code. The current implementation uses around 44 such transformations (i.e., $n = 44$). Combining several simple transformations improves the decoupling among different language constructs and facilitates reuse (e.g., to implement a translator of C to Java) and debugging: the intermediate programs are easily understandable by programmers familiar with both C and Eiffel.

Helper classes. The core of the translation from C to Eiffel must ensure that Eiffel applications have access to objects with the same capabilities as their counterparts in C; for example, an Eiffel class that translates a C **struct** has to support field access and every other operation defined on **struct**s. Conversely, C *external* pre-compiled code may also have to access the Eiffel representations of C constructs; for example, the Eiffel translation of a C

program calling *printf* to print a local string variable *str* of type **char**$*$ must grant *printf* access to the Eiffel object that translates *str*, in conformance with C's conventions on strings. To meet these requirements, C2Eif includes a limited number of hand-written helper Eiffel classes that bridge the Eiffel and C environments; their names are prefixed by *CE* for C *and* Eiffel. Rather than directly replicating or wrapping commonly used external libraries (such as `stdio` and `stdlib`), the helper classes target C fundamental *language features* and in particular types and type constructors. This approach works with any external library, even non-standard ones, and is easier to maintain because it involves only a limited number of classes. We now give a concise description of the most important helper classes; Section 3.3 shows detailed examples of their usage.

- *CE_POINTER* [*G*] represents C pointers of any type through the generic parameter *G*. It includes features to perform full-fledged pointer arithmetic and to get pointer representations that C can access but the Eiffel's runtime will not modify (in particular, the garbage collector will not modify pointed addresses nor relocate memory areas).
- *CE_CLASS* defines the basic interface of Eiffel classes that correspond to C **union**s and **struct**s. It includes features that return instances of class *CE_POINTER* pointing to a memory representation of the structure that C can access.
- *CE_ARRAY* [*G*] extends *CE_POINTER* and provides consistent array access to both C and Eiffel (according to their respective conventions). It includes contracts that check for out-of-bound access.
- *CE_ROUTINE* represents function pointers. It supports calls to Eiffel routines through **agent**s—Eiffel's construct for function objects (*closures* or *delegates* in other languages)—and calls to (and callbacks from) external C functions through raw function pointers.
- *CE_VA_LIST* supports variadic functions, using Eiffel class *TUPLE* (sequences of elements of heterogeneous type) to store a variable number of arguments. It offers an Eiffel interface that extends the standard C's (declared in `stdarg.h`), as well as output in a format accessible by external C code.

3.3 Translating C to Eiffel

This section presents the major components of the translation $\mathcal{T}$ from C to Eiffel implemented in C2Eif, and illustrates the general rules with a number

of small examples. The presentation breaks $\mathcal{T}$ down into several components that target different language aspects (for example, $\mathcal{T}_{\text{TD}}$ maps C type declarations to Eiffel classes); these components mirror the incremental transformations T_i of C2Eif (mentioned in Section 3.2) but occasionally overlook inessential details for greater presentation clarity.

Section 3.3.6 then gives a short overview of all actual translation steps as currently implemented in the supporting tool C2Eif.

External functions in Eiffel. Eiffel code translated from C normally includes calls to external C pre-compiled functions, whose actual arguments correspond to objects in the Eiffel runtime. This feature relies on the **external** Eiffel language construct: Eiffel routines can be declared as **external** and directly execute C code embedded as Eiffel strings[2] or call functions declared in header files. For example, the following Eiffel routine *sin_twice* returns twice the sine of its argument by calling the C library function *sin* (declared in `math.h`):

```
sin_twice (arg: REAL_32): REAL_32
    external
        C inline use <math.h>
    alias
        return 2 * sin($arg);
    end
```

Calls using **external** can exchange arguments between the Eiffel and the C runtimes only for a limited set of primitive type: numeric types (that have the same underlying machine representation in Eiffel and C) and instances of the Eiffel system class *POINTER* that corresponds to raw untyped C pointers (not germane to Eiffel's pointer representation, unlike *CE_POINTER*). In the *sin_twice* example, argument *arg* of numeric type *REAL_32* is passed to the C runtime as `$arg`. Every helper class (described in Section 3.2) includes an attribute *c_pointer* of type *POINTER* that offers access to a C-conforming representation usable in **external** calls.

3.3.1 Types and Type Constructors

C declarations $T\,v$ of a variable v of type T become Eiffel declarations $v{:}\mathcal{T}_{\text{TY}}(T)$, where $\mathcal{T}_{\text{TY}}$ is the mapping from C types to Eiffel classes described in this section.

Numeric types. C numeric types correspond to Eiffel classes *INTEGER* (signed integers), *NATURAL* (unsigned integers), *REAL* (floating point num-

[2]For readability, we will omit quotes in **external** strings.

bers) with the appropriate bit-size as follows[3]:

C TYPE T	EIFFEL CLASS $\mathcal{T}_{\text{TY}}(T)$
char	*INTEGER_8*
short int	*INTEGER_16*
int, **long int**	*INTEGER_32*
long long int	*INTEGER_64*
float	*REAL_32*
double	*REAL_64*
long double	*REAL_96*

Unsigned variants follow the same size conventions as signed integers but for class *NATURAL*; for example $\mathcal{T}_{\text{TY}}$(**unsigned short int**) is *NATURAL_16*.

Pointers. Pointer types are translated using class *CE_POINTER* [*G*] with the generic parameter *G* instantiated with the pointed type:

$$\mathcal{T}_{\text{TY}}(T\,*) \;=\; \textit{CE\_POINTER}\,[\mathcal{T}_{\text{TY}}(T)]$$

with the convention that $\mathcal{T}_{\text{TY}}$(**void**) maps to Eiffel class *ANY*, ancestor to every other class (*Object* in Java). The definition works recursively for multiple indirections; for example, *CE_POINTER*[*CE_POINTER*[*REAL_32*]] stands for $\mathcal{T}_{\text{TY}}$(**float** $**$).

Function pointers. Function pointers are translated to Eiffel using class *CE_ROUTINE*:

$$\mathcal{T}_{\text{TY}}(T_0\ (*)\ (T_1,\ ...,T_n)) \;=\; \textit{CE\_ROUTINE}$$

CE_ROUTINE inherits from *CE_POINTER* [*ANY*], hence it behaves as a generic pointer, but it specializes it with references to **agent**s that wrap the functions pointed to; Section 3.3.2 describes this mechanism.

Arrays. Array types are translated to Eiffel using class *CE_ARRAY*[*G*] with the generic parameter *G* instantiated with the array base type: $\mathcal{T}_{\text{TY}}(T\,[n])$ = *CE_ARRAY*[$\mathcal{T}_{\text{TY}}(T)$]. The size parameter n, if present, does not affect the declaration, but initializations of array variables use it (see Section 3.3.2). Multi-dimensional arrays are defined recursively as arrays of arrays: $\mathcal{T}_{\text{TY}}(T\,[n_1][n_2]...[n_m])$ is then *CE_ARRAY*[$\mathcal{T}_{\text{TY}}(T[n_2]...[n_m])$].

Enumerations. For every **enum** type *E* defined or used, the translation introduces an Eiffel class *E* defined by the translation $\mathcal{T}_{\text{TD}}$ (for type definition):

[3]We implemented class *REAL_96* specifically to support **long double** on Linux machines.

$\mathcal{T}_{\mathrm{TD}}(\textbf{enum}\ E\,\{v_1 = k_1, \ldots, v_m = k_m\}) = \textbf{class}\ E\ \textbf{feature}$
v_1: *INTEGER_32* $=k_1$; ... ; v_m: *INTEGER_32* $=k_m$ **end**

Class E has as many attributes as the **enum** type has values, and each attribute is an integer that receives the corresponding value in the enumeration. Every C variable of type E also becomes an integer variable in Eiffel (that is, $\mathcal{T}_{\mathrm{TY}}(\textbf{enum}\ E) = \mathit{INTEGER\_32}$), and class E is only used to assign constant values according to the **enum** naming scheme.

Structs and unions. For every compound **struct** type S defined or used, the translation introduces an Eiffel class S:

class S **inherit** *CE_CLASS* **feature** $\mathcal{T}_{\mathrm{F}}(T_1\ v_1) \ldots \mathcal{T}_{\mathrm{F}}(T_m\ v_m)$ **end**

for $\mathcal{T}_{\mathrm{TD}}(\textbf{struct}\ S\,\{T_1\ v_1; \ldots; T_m\ v_m\})$. Correspondingly, $\mathcal{T}_{\mathrm{TY}}(S) = S$; that is, variables of type S become references of class S in Eiffel. The translation $\mathcal{T}_{\mathrm{F}}(T\ v)$ of each field v of the **struct** S introduces an attribute of the appropriate type in class S, and a setter routine *set_v* that also updates the underlying C representation of v:

```
v: T_TY(T) assign set_v   -- declares 'set_v' as the setter of v

set_v (a_v: T_TY(T))
    do
        v := a_v
        update_memory_field ("v")
    end
```

Class *CE_CLASS*, of which S is a subclass, implements *update_memory_field* using reflection, so that the underlying C representation is created and updated dynamically only when needed during execution (for example, to pass a **struct** instance to a native C library), thus avoiding any data duplication overhead whenever possible.

The translation of **union** types follows the same lines as that of **struct**s, with the only difference that classes translating **union**s generate the underlying C representation in any case upon initialization, even if the **union** is not passed to the C runtime; calls to *update_memory_field* update all attributes of the class to reflect the correct memory value. We found this to be a reasonable compromise between performance and complexity of memory management of **union** types where, unlike **struct**s, fields share the same memory space.

Example 1 *Consider a C* ***struct*** *car that contains an integer field* plate_num *and a string field* brand*:*

```
typedef struct {
    unsigned int plate_num;
    char *brand;
} car;
```

The translation $\mathcal{T}_{\text{TD}}$ introduces a class CAR as follows:

```
class
    CAR

inherit
    CE_CLASS

feature

    plate_num: NATURAL_32 assign set_plate_num

    brand: CE_POINTER [INTEGER_8] assign set_brand

    set_plate_num (a_plate_num: NATURAL_32)
        do
            plate_num := a_plate_num
            update_memory_field ("plate_num")
        end

    set_brand (a_brand: CE_POINTER [INTEGER_8])
        do
            brand := a_brand
            update_memory_field ("brand")
        end

end
```

3.3.2 Variable Initialization and Usage

Initialization. Eiffel variable declarations v: C only allocate memory for a *reference* to objects of class C, and initialize it to **Void** (**null** in Java). The only exceptions are, once again, numeric types: a declaration such as n: *INTEGER_64* reserves memory for a 64-bit integer and initializes it to zero. Therefore, every C local variable declaration $T\ v$ of a variable v of type T may also produce an *initialization*, consisting of calls to creation procedures (constructors in Java) of the corresponding helper classes, as specified by the

declaration mapping $\mathcal{T}_{\mathrm{DE}}$:

$$\mathcal{T}_{\mathrm{DE}}(T\ v;) = \begin{cases} v : \mathcal{T}_{\mathrm{TY}}(T) & \text{(NT)} \\ v : \mathcal{T}_{\mathrm{TY}}(T);\ \textbf{create}\ v.make(\ll n_1, \ldots, n_m \gg) & \text{(AT)} \\ v : \mathcal{T}_{\mathrm{TY}}(T);\ \textbf{create}\ v.make & \text{(OT)} \end{cases}$$

where definition (NT) applies if T is a numeric type; (AT) applies if T is an array type $S[n_1],\ldots,[n_m]$; and (OT) applies otherwise. The creation procedure *make* of *CE_ARRAY* takes a sequence of integer values to allocate the right amount of memory for each array dimension; for example **int** *a*[2][3] is initialized by **create** *a.make*(≪2, 3≫).

Memory management. Helper classes are regular Eiffel classes, therefore the Eiffel garbage collector disposes instances when they are no longer referenced (for example, when a local variable gets out of scope). Upon collection, the *dispose* finalizer routines of the helper classes ensure that the C memory representations are also appropriately deallocated; for example, the finalizer of *CE_ARRAY* frees the array memory area by calling *free* on the attribute *c_pointer*.

To replicate the usage of *malloc* and *free*, we offer wrapper routines that emulate the syntax and functionalities of their C homonym functions, but operate on *CE_POINTER*: they get raw C pointers by external calls to C library functions, convert them to *CE_POINTER*, and record the dynamic information about allocated memory size. The latter is used to check that successive usages conform to the declared size (see Section 3.4.4). Finally, the creation procedure *make_cast* of the helper classes can convert a generic pointer returned by *malloc* to the proper pointed type, according to the following translation scheme:

C code	Translated Eiffel code
$T*\ p$;	p: *CE_POINTER*[$\mathcal{T}_{\mathrm{TY}}(T)$]
$p = (T*)malloc(\textbf{sizeof}(T))$;	**create** *p.make_cast* (*malloc* ($\sigma(T)$))
free(p);	*free*(p)

where σ is an encoding of the size information.

Variable usage. The translation of variable usage is straightforward: variable reads in expressions are replicated verbatim, and C assignments (=) become Eiffel assignments (:=); the latter is, for classes translating C **struct**s and **union**s, *CE_ARRAY*, and *CE_POINTER*, syntactic sugar for calls to setter routines that achieve the desired effect. The only exceptions occur when implicit type conversions in C must become explicit in Eiffel, which may spoil the readability of the translated code but is necessary with strong typing. For example, the C assignment *cr* = **'s'**—assigning character constant **'s'** to variable *cr* of type **char**—becomes the Eiffel assignment

cr := (**'s'**).*code.to_integer_8* that encodes **'s'** with the proper representation.

Variable address. Whenever the address &*v* of a C variable *v* of type *T* is taken, *v* is translated as an array of unit size and type *T*: $\mathcal{T}_{\mathrm{DE}}(T\ v) = \mathcal{T}_{\mathrm{DE}}(T\ v[1])$, and every usage of *v* is adapted accordingly: &*v* becomes just *v*, and occurrences of *v* in expressions become $*v$. This little hack makes it possible to have Eiffel assignments translate C assignment uniformly; otherwise, usages of *v* should have different translations according to whether the information about *v*'s memory location is copied around (with &) or not.

Dereferencing, pointer arithmetic. The helper class *CE_POINTER* features a query *item* that translates dereferencing ($*$) of C pointers. Pointer arithmetic is translated verbatim, because class *CE_POINTER* overloads the arithmetic operators to be aliases of proper underlying pointer manipulations, so that an expression such as $p + 3$ in Eiffel, for references *p* of type *CE_POINTER*, hides the explicit expression *c_pointer* + 3 $*$ *element_size*.

Example 2 *Consider a double pointer variable* carsp *with target type* car *(defined in Example 1), and an integer variable* num*. The following C code fragment declares the variables, reserves space for* carsp *to point to an array of* num *consecutive cars, and lets another variable* p *point to the third element of the array.*

```
car **carsp;
int num;
car *p;
*carsp = malloc(num * sizeof(car));
p = *carsp + 2;
```

The Eiffel translation of the code fragment is as follows (notice the implicit conversions that become explicit in the calls).

```
carsp: CE_POINTER [CE_POINTER [CAR]]
num: INTEGER_32
p: CE_POINTER [CAR]
carsp.item := create {CE_POINTER [CAR]}.make_cast (malloc (
        num.to_natural_32 *
        (create {CAR}.make).structure_size.to_natural_32
    ))
p := carsp.item + 2
```

Using function pointers. Class *CE_ROUTINE*, which translates C function pointers, is usable both in the Eiffel and in the C environment (see Figure 3.2). On the Eiffel side, its instances wrap Eiffel routines using *agents*—Eiffel's mechanism for function objects. A private attribute

routine references objects of type *ROUTINE* [*ANY*, *TUPLE*], an Eiffel system class that corresponds to agents wrapping routines with any number of arguments and argument types stored in a tuple. Thus, Eiffel code can use the **agent** mechanism to create instances of class *ROUTINE*. For example, if *foo* denotes a routine of the current class and *fp* has type *CE_ROUTINE*, **create** *fp.make_agent* (**agent** *foo*) makes *fp*'s attribute *routine* point to *foo*. On the C side, when function pointers are directly created from C pointers (e.g., references to external C functions), *CE_ROUTINE* behaves as a wrapper of raw C function pointers, and dynamically creates invocations to the pointed functions using the library `libffi` [12].

The Eiffel interface to *CE_ROUTINE* will then translate calls to wrapped functions into either **agent** invocations or external calls with `libffi` according to how the class has been instantiated. Assume, for example, that *fp* is an object of class *CE_ROUTINE* that wraps a procedure with one integer argument. If *fp* has been created with an Eiffel **agent** *foo* as above, calling *fp.call* ([42]) wraps the call *foo* (42) (edge 1 in Figure 3.2); if, instead, *fp* only maintains a raw C function pointer, the same instruction *fp.call* ([42]) creates a native C call using `libffi` (edge 2 in Figure 3.2).

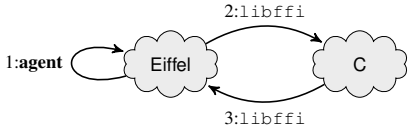

Figure 3.2: Calls through function pointers.

The services of class *CE_ROUTINE* behave as an adapter between the procedural and object-oriented representation of routines: the signatures of C functions must change when they are translated to Eiffel routines, because routines in object-oriented languages include references to a target object as implicit first argument. Calls from external C code to Eiffel routines are therefore intercepted at runtime with `libffi` callbacks (edge 3 in Figure 3.2) and dynamically converted to suitable **agent** invocations.

3.3.3 Control Flow

This section discusses the translation $\mathcal{T}_{\text{CF}}$ of instructions directing the control flow. **Sequential** composition, **conditionals**, and **loops** are quite similar in C and Eiffel:

$$
\begin{array}{ll}
\mathcal{T}_{\mathrm{CF}}(I_1\,;\,I_2) & = \mathcal{T}(I_1)\,;\,\mathcal{T}(I_2) \\
\mathcal{T}_{\mathrm{CF}}(\textbf{if}\ (c)\,\{TB\}\ \textbf{else}\ \{EB\}) & = \textbf{if}\ \mathcal{T}(c) \\
 & \quad \textbf{then}\ \mathcal{T}(TB)\ \textbf{else}\ \mathcal{T}(EB)\ \textbf{end} \\
\mathcal{T}_{\mathrm{CF}}(\textbf{while}\ (c)\,\{LB\}) & = \textbf{from until not}\ \mathcal{T}(c) \\
 & \quad \textbf{loop}\ \mathcal{T}(LB)\ \textbf{end}
\end{array}
$$

Jumps. Eiffel enforces structured programming, hence it lacks control-flow breaking instructions such as C's **goto**, **break**, **continue**, **return**. The translation $\mathcal{T}_{\mathrm{CF}}$ eliminates them along the lines of the *global* version—using Harel's terminology [16]—of the structured programming theorem. Every C function *foo* using **goto** determines a list of instructions $s_0, s_1, \ldots, s_n$, where each s_i is a maximal sequential block of instructions, with no labels after the first instruction or jumps before the last one. $\mathcal{T}_{\mathrm{CF}}$ translates *foo*'s body into a single loop over an auxiliary integer variable *pc* that emulates a *program counter*:

$$
\mathcal{T}_{\mathrm{CF}}(\langle s_0, s_1, \ldots, s_n \rangle) = \left\{
\begin{array}{l}
\textbf{from}\ pc := 0\ \textbf{until}\ pc = -1\ \textbf{loop} \\
\ \textbf{inspect}\ pc \\
\ \ \textbf{when}\ 0\ \textbf{then}\ \mathcal{T}(s_0)\,;\,upd(pc) \\
\ \ \textbf{when}\ 1\ \textbf{then}\ \mathcal{T}(s_1)\,;\,upd(pc) \\
\ \ \vdots \\
\ \ \textbf{when}\ n\ \textbf{then}\ \mathcal{T}(s_n)\,;\,upd(pc) \\
\ \textbf{end} \\
\textbf{end}
\end{array}
\right.
$$

pc is initially zero; every iteration of the loop body executes s_{pc} for the current value of *pc*, and then updates *pc* ($upd(pc)$) to determine the next instruction to be executed: blocks ending with jumps modify *pc* directly, other blocks increment it by one, and exit blocks set it to -1, which makes the overall loop terminate (whenever the original function terminates).

This translation supports all control-flow breaking instructions, and in particular **continue**, **break**, and **return**, which are special cases of **goto**. $\mathcal{T}_{\mathrm{CF}}$, however, improves the readability in these special cases by directly using auxiliary Boolean flag variables (with the same names as the instruction they replace) that are tested in the translated exit conditions and achieve the same effect with smaller changes to the code structure.
For example,

```
while (n > 0) {
    if (n == 3)  break;
    n--;
}
```

becomes:

from
until
  break **or not** $n > 0$
loop
  if $n = 3$ **then**
    break := **True**
  end
  if not *break* **then**
    $n := n - 1$
  end
end

Functions. Function definitions and calls directly translate to routine definitions and calls in Eiffel:

$$\mathcal{T}_{\mathrm{CF}}(T_0\ \mathit{foo}\ (T_1 a_1, \ldots, T_n a_n)\{\ B\ \}) =$$
$$\mathit{foo}(a_1 : \mathcal{T}_{\mathrm{TY}}(T_1); \ldots; a_n : \mathcal{T}_{\mathrm{TY}}(T_n)) : \mathcal{T}_{\mathrm{TY}}(T_0)\ \mathbf{do}\ \mathcal{T}(B)\ \mathbf{end}$$
$$\mathcal{T}_{\mathrm{CF}}(\mathit{foo}\ (e_1, \ldots, e_n)) = \mathit{foo}\ (\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n))$$

When T_0 is **void**, ": $\mathcal{T}_{\mathrm{TY}}(T_0)$" is omitted in Eiffel.

Translations of variadic function definitions use the Eiffel *TUPLE* class:

$$\mathcal{T}_{\mathrm{CF}}(T_0\ \mathit{var}\ (T_1 a_1, \ldots, T_n a_n, \ldots)) =$$
$$\mathit{var}\ (\text{args: TUPLE}[a_1 : \mathcal{T}_{\mathrm{TY}}(T_1); \ldots; a_n : \mathcal{T}_{\mathrm{TY}}(T_n)]) : \mathcal{T}_{\mathrm{TY}}(T_0)$$

Eiffel's type system prescribes that every $(n + m)$-*TUPLE* with types $[T_1, \ldots, T_n, T_{n+1}, \ldots, T_{n+m}]$, $m \geq 0$, conforms to any shorter n-*TUPLE* with $[T_1, \ldots, T_n]$. Therefore, calls to variadic functions can use longer tuples to accommodate the additional optional arguments:

$$\mathcal{T}_{\mathrm{CF}}(\mathit{var}\ (e_1, \ldots, e_n, e_{n+1}, \ldots, e_{n+m})) =$$
$$\mathit{var}\ ([\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n), \mathcal{T}(e_{n+1}), \ldots, \mathcal{T}(e_{n+m})])$$

We can access arguments in a *TUPLE* either with standard Eiffel syntax, or using the helper class *CE_ VA_ LIST*. With standard syntax, $\mathit{args}.a_i$ refers to the required argument with name a_i, $1 \leq i \leq n$, and $\mathit{args}.\mathcal{T}_{\mathrm{TY}}(T_k)\_\mathit{item}(k)$ refers to the k-th argument a_k, for any $1 \leq k \leq n$ (required) or $k > n$ (optional), where $\mathcal{T}_{\mathrm{TY}}(T_k)$ is a_k's type. Alternatively, *CE_ VA_ LIST* provides a uniform interface to both Eiffel and C that accesses argument lists sequentially, replicating and interoperable with `stdarg`'s:

argp: *CE_ VA_ LIST*
create *argp.make* (*args*, $n{+}1$)
$e_1 :=$ $\mathit{argp}.\mathcal{T}_{\mathrm{TY}}(T_{n+1})\_\mathit{item}$ −− first optional argument a_{n+1}
$e_2 :=$ $\mathit{argp}.\mathcal{T}_{\mathrm{TY}}(T_{n+2})\_\mathit{item}$ −− second optional argument a_{n+2}
...

Continuing the running Examples 1–2, consider a function *init_cars* that takes a pointer *carsp* to an array of cars, an integer *num*, and *num* pairs of integers and strings, and initializes the array pointed by *carsp* with *num* cars with the plate numbers and brands given in the optional arguments.

```
void init_cars(car **carsp, int num, ...) {
    va_list argp;
    car *ccar;
    int n;
    *carsp = malloc(num * sizeof(car));
    ccar = *carsp;
    va_start(argp, num);
    for(n = num ; n > 0; n--) {
        ccar->plate_num = va_arg(argp, unsigned int);
        ccar->brand = va_arg(argp, char*);
        ccar++;
    }
}
```

The translation $\mathcal{T}$ of *init_cars* is as follows.

```
init_cars (args: TUPLE [
        carsp: CE_POINTER [CE_POINTER [CAR]];
        num: INTEGER_32])
    local
        argp: CE_VA_LIST
        ccar: CE_POINTER [CAR]
        n: INTEGER_32
    do
        args.carsp.item := create {CE_POINTER [CAR]}.make_cast (
                malloc (
                        num.to_natural_32 *
                        (create {CAR}.make).structure_size.to_natural_32
                    ))
        ccar := args.carsp.item
        create argp.make (args, 3)
        n := num
        from until not n >0 loop
            ccar.item.plate_num := argp.natural_32_item
            ccar.item.brand := (create {CE_POINTER [INTEGER_8]}.
                make_cast (argp.pointer_item))
            ccar := ccar + 1
            n := n - 1
        end
    end
```

Long jumps. The C library `setjmp` provides the functions *setjmp* and *longjmp* to save an arbitrary return point and jump back to it across function call boundaries. The wrapping mechanism used for external functions (see Section 3.3.4) does not work to replicate long jumps, because the return values saved by *setjmp* wrapped as an external function are no longer valid after execution leaves the wrapper. Therefore, C2Eif translates *setjmp* and *longjmp* by means of the helper class *CE_EXCEPTION*. As the name suggests, *CE_EXCEPTION* uses Eiffel's exception propagation mechanism to go back in the call stack to the allocation frame of the function that called *setjmp*. There, translated **goto** instructions jump to the specific point saved with *setjmp* within the function body.

3.3.4 Object-Oriented Encapsulation

Externals. For every included system header `header.h`, $\mathcal{T}$ defines a class *HEADER* with wrappers for all external functions and variables declared in `header.h`. The wrappers are routines using the Eiffel **external** mechanism and performing the necessary conversions between the Eiffel and the C runtimes. In particular, external functions using only numeric types, which are interoperable between C and Eiffel, directly map to wrapper routines; for example, *exit* in `stdlib.h` is:

```
exit (status: INTEGER_32)
    external
        C inline use <stdlib.h>
    alias
        exit($status);
    end
```

When external functions involve types using helper classes in Eiffel, a routine passes the underlying C representation to the external calls; for example, *fclose* in `stdio.h` generates:

```
fclose (stream: CE_POINTER [ANY]): INTEGER_32
    do
        Result := c_fclose (stream.c_pointer)
    end
c_fclose (stream: POINTER): INTEGER_32
    external
        C inline use <stdio.h>
    alias
        return fclose($stream);
    end
```

In some complex cases—typically, with variadic external functions—the wrapper can only assemble the actual call on the fly at runtime. This is done using *CE_ROUTINE*; for example, *printf* is wrapped as:

```
printf (args: TUPLE [format: CE_POINTER[INTEGER_8]]):
    INTEGER_32
  do
    Result := (create {CE_ROUTINE}.make_shared (c_printf)).
        integer_32_item (args)
  end

c_printf: POINTER
  external
    C inline use <stdio.h>
  alias
    return &printf;
  end
```

The translation can also inline **assembly code**, using the same mechanisms as external function calls.

Globals. For every source file `source.c`, $\mathcal{T}$ defines a class *SOURCE* that includes translations of all function definitions (as routines) and global variables (as attributes) in `source.c`. Class *SOURCE* also *inherits* from the translation of each system header file that `source.c` includes, in order to gain access to required external functions and variables. For example, if `foo.c` includes `stdio.h`, *FOO* is declared as **class** *FOO* **inherit** *STDIO*.

3.3.5 Formatted Output Optimization

Library function *printf* is the standard C output function, which prints values according to format strings passed as input. Eiffel offers the command *Io.put_string* to print plain strings to standard output, and type-specific formatter classes (e.g., *FORMAT_INTEGER*) to convert numeric types into formatted strings. With the goal of making the translated code as close as possible to standard Eiffel, C2Eif replaces calls to *printf* with equivalent invocations of *Io.put_string* and the formatters when possible: whenever *printf*'s returned value is not used, and the format string is a constant literal (or only one argument is passed to *printf*, hence the string is meant to be interpreted verbatim). In these situations, C2Eif parses literal format strings to translate them into equivalent calls to *Io.put_string*. If this process finds a mismatch between a format specifier and the type of the corresponding argument to *printf*, it issues a warning. In these cases, and in all other situations where

direct usage of *put_string* is not possible, C2Eif falls back to use wrapped calls to the external *printf* function. The tool supports the same optimizations for the variants of *printf* such as *fprintf*.

3.3.6 Overview of the Translation Steps

In this section we give an overview of all translation steps as currently implemented in the supporting tool C2Eif. Each step is implemented in a separate class; below we give the class name, the description of its purpose and a simple example, where appropriate. While most of the steps perform successive incremental translations on the AST, a few compute information required in later steps.

The order of the steps was defined in an "ad hoc" manner during the development of C2Eif. The basic principle is to apply simplifying and unifying transformations (e.g. removing typedefs or adapting increment and decrement) as early as possible, so that the following steps don't have to deal with special cases. The detailed order, however, was subject to constant adaptation.

A theoretical foundation for incremental transformation ordering would be of great value since this is a common problem I have encountered not only in this thesis with C2Eif and J2Eif, but also in projects in industry.

1. *SOURCE_LOADER* creates an AST from the provided CIL C file.

2. *TYPEDEF_REMOVER* removes **typedef** declarations and replaces **typedef** types with their basic C types:

   typedef unsigned int *size_t*;
   size_t count; → **unsigned int** *count*;

3. *ENUM_TYPE_TRANSFORMER* changes declarations with enum types to use the corresponding C basic type (**int** or **unsigned int**) and replaces number literals cast to an enum type (mostly introduced by CIL) with the corresponding enum field name:

C	CIL	C2Eif
enum *day d* = *saturday*;	**enum** *day d* = (**enum** *day*)0;	**unsigned int** *d* = *saturday*;

   See Section 3.3.1.

4. *FORWARD_DECLARATIONS_REMOVER* removes all forward declarations.

5. *BINDING_ADDER* computes declaration and type bindings for expressions.

6. *INCREMENT_ DECREMENT_ TRANSFORMER* changes increment and decrement expressions to the corresponding regular assignments (CIL moved expressions with side-effects to independent statements):

$$i{+}{+}; \quad \rightarrow \quad i = i + 1;$$

7. *SIZEOF_ TRANSFORMER* transforms **sizeof**() expressions to the equivalent Eiffel counterparts:

$$\textbf{sizeof(int)} \quad \rightarrow \quad \{PLATFORM\}.Integer\_32\_bytes$$

8. *FLUSH_ TRANSFORMER* transforms *flush*() calls to the Eiffel standard library counterparts:

$$\begin{array}{l} fflush(stdout); \\ fflush(stderr); \end{array} \quad \rightarrow \quad \begin{array}{l} Io.standard\_default.flush \\ Io.error.flush \end{array}$$

See Section 3.4.3.

9. *CIL_ CHAR_ TRANSFORMER* reverts CILs **char**[] initialization adaptions to use regular string literals:

C	CIL	C2Eif
char *s*[] = **"ab"**;	**char** *s*[3] = { (**char**)**'a'**, (**char**)**'b'**, (**char**)**'\000'** };	**char** *s*[3] = **"ab\000"**;

10. *CIL_ POINTER_ ACCESS_ TRANSFORMER* reverts CILs pointer access adaptions to more readable array style accesses:

C	CIL	C2Eif
s[1]	$*(s + 1)$	*s*[1]

11. *ASSIGNMENT_ TRANSFORMER* transforms all assignment variants to the standard assignment:

$$i *= 3; \quad \rightarrow \quad i = i * 3;$$

12. *POINTER_ ACCESS_ INDEX_ TRANSFORMER* ensures that the index for pointer accesses is of type **int**:

$$\begin{array}{l} \textbf{unsigned int } ui = 1; \\ \textbf{char } *s = \textbf{"abc"}; \\ printf(\textbf{"\%c\textbackslash n"}, s[ui]); \end{array} \quad \rightarrow \quad \begin{array}{l} \textbf{unsigned int } ui = 1; \\ \textbf{char } *s = \textbf{"abc"}; \\ printf(\textbf{"\%c\textbackslash n"}, s[(\textbf{int})ui]); \end{array}$$

13. *ARITHMETIC_ OPERATION_ TRANSFORMER* adapts arithmetic operations not directly supported by Eiffel to supported ones:

$$\begin{array}{l} \textbf{unsigned int } ui = 1; \\ ui = -ui; \end{array} \quad \rightarrow \quad \begin{array}{l} \textbf{unsigned int } ui = 1; \\ ui = (\textbf{unsigned int}) - (\textbf{int})ui; \end{array}$$

14. *CAST_ REMOVER* removes unnecessary casts (mostly introduced by CIL):

C	CIL	C2Eif
char $*s =$ **"abc"**;	**char** $*s =$ (**char** $*$)**"abc"**;	**char** $*s =$ **"abc"**;

15. *POINTER_ TRANSFORMER* adapts pointer operations to feature calls as required in the Eiffel code:

void $*p1$ = ...;
void $*p2$ = ...;
bool $b = (p1 == p2)$;
$\rightarrow$
void $*p1$ = ...;
void $*p2$ = ...;
bool $b = (p1.is\_equal\ (p2))$;

16. *CAST_ TRANSFORMER* transforms C casts to Eiffel casts:

(**int**)1.2 $\rightarrow$ (1.2).*truncated_ to_ integer*

17. *PRINTF_ TRANSFORMER* transforms *printf*() calls to the Eiffel library conterparts where possible:

printf(**"Version: %d\n"**, *VERSION*); $\rightarrow$ *Io.put_ string* (**"Version: "** + *VERSION.out* + **"\n"**);

See Section 3.3.5.

18. *STD_ STREAM_ TRANSFORMER* replaces *stdin*, *stdout* and *stderr* with their Eiffel equivalent:

stdin, *stdout*, *stderr* $\rightarrow$ *Io.input.file_ pointer*, *Io.output.file_ pointer*, *Io.error.file_ pointer*

19. *LITERAL_ TRANSFORMER* adapts C literals to Eiffel literals where necessary:

"\t\r\n" $\rightarrow$ "%T%R%N"

20. *BOOLEAN_ TRANSFORMER* transforms all conditional expressions into boolean expressions:

if(1){ ... } $\rightarrow$ **if**((1).*to_ boolean*){ ... }

21. *PARAMETER_ ASSIGNMENT_ TRANSFORMER* removes assignments to formal parameters, using auxiliary local variables instead:

void *foo*(**int** *i*) {
  $i = 1$;
}
$\rightarrow$
void *foo*(**int** *a_ i*) {
  int *l_ i* = *a_ i*;
  l_ i = 1;
}

22. *ASSEMBLER_ TRANSFORMER* moves inline assembly code into separate functions (in Eiffel they become inline C code with inline assembly) and adds calls to them:

```
void foo() {
  int src = 123;
  int dest;
  asm ("mov %1, %0":
       "=r" (dest) :
       "r" (src));
}
```

→

```
void foo() {
  int src = 123;
  int dest;
  asm1(&dest, &src);
}

void asm1(int *dest_p, int *src_p) {
  int dest = *dest_p;
  int src = *src_p;

  asm ("mov %1, %0" :
       "=r" (dest) :
       "r" (src));

  *dest_p = dest;
  *src_p = src;
}
```

23. *VARIABLE_ ADDRESS_ TRANSFORMER* turns variables, whose address are taken, to one-element arrays:

```
void foo() {
  int i = 23;
  int *ip = &i;
  printf("%d %d", i, *ip);
}
```

→

```
void foo() {
  int i[1] = {23};
  int *ip = i;
  Io.put_string ((*i).out + " " +
                 (*ip).out);
}
```

See Section 3.3.2.

24. *ADDRESS_ OPERATOR_ TRANSFORMER* adapts the address-of operator in cases that are not directly supported in Eiffel:

$$\&foo[bar] \quad \rightarrow \quad foo + bar$$

25. *BREAK_ CONTINUE_ RETURN_ TRANSFORMER* transforms the **break**, **continue** and **return** statements to equivalent code using a combination of boolean flags and conditional statements. See Section 3.3.3.

26. *LONGJMP_ TRANSFORMER* transforms *setjmp*() and *longjmp*() to a combination of exceptions and **goto**s. See Section 3.3.3.

27. *SWITCH_ TRANSFORMER* transforms C **switch** statements to Eiffel **inspect** statements with **goto**s:

```
switch(i) {                          →   inspect i
  case 1: printf("1");                     when 1 then
  case 2: printf("2"); break;              when 2 then
  default: printf("3");                    else
}                                          end

                                           state1:
                                             Io.put_string ("1");
                                           state2:
                                             Io.put_string ("2");
                                           state3:
                                             Io.put_string ("3");
                                           state4:
```

28. *GOTO_TRANSFORMER* transforms **goto** statements to equivalent Eiffel code. See Section 3.3.3.

29. *VARARG_TRANSFORMER* transforms functions with variable arguments into routines taking one *TUPLE* argument. See Section 3.3.3.

30. *ALLOCA_TRANSFORMER* transforms *alloca*() calls to use a helper class provided by C2Eif.

31. *BUILTIN_TRANSFORMER* handles some exotic GCC builtin functions. E.g. removing *__builtin_prefetch* calls.

32. *EXPRESSION_AS_INSTRUCTION_TRANSFORMER* adds a call to *do_nothing* at the end of every expression used as an instruction:

```
printf(s1, s2);   →   printf(s1, s2).do_nothing;
```

33. *EXPANDED_TRANSFORMER* restores expanded (copy-by-value) behavior of **struct**s and *va_list*s by cloning them where necessary; this is required since **struct**s and *va_list*s are translated into reference types:

```
struct person p1 = {"Taco", 30};   →   struct person p1 = {"Taco", 30};
struct person p2 = p1;                 struct person p2 = p1.twin;
```

34. *CFG_CREATOR* computes the control flow graph for the original C code.

35. *EIFFEL_CFG_CREATOR* computes the control flow graph for the Eiffel code (which might be different due to transformations of jump statements).

36. *CALL_GRAPH_CREATOR* computes the call graph.

37. *UNUSED_SYSTEM_FEATURES_REMOVER* removes unused system features (found by analyzing the call graph).

38. *CLASSES_START* prepares the source code for the creation of classes; sets class prefixes to avoid conflicting names.

39. *CLASS_CREATOR* creates classes based on the original C source files. See Section 3.3.4.

40. *CLASSES_FINISH* finishes up class creation; adds superclasses based on features used from the system interface. See Section 3.3.4.

41. *CIL_TEMP_VARIABLE_REMOVER* removes temporary variables introduced by CIL which were not needed in C2Eif transformations:

C	CIL	C2Eif
printf(**"%d\n"**, *printf*(**"a"**));	**int** *tmp* = *printf*(**"a"**); *printf*(**"%d\n"**, *tmp*);	*Io.put_string* (*printf*(**"a"**).*out* + **"%N"**);

42. *IDENTIFIER_TRANSFORMER* transforms names of fields, variables and functions to adhere to the Eiffel namespace and keyword constraints.

43. *INLINER* allows functions to be inlined as C source code instead of being translated to Eiffel (if this feature is activated in C2Eif).

44. *EIFFEL_FILE_WRITER* creates the Eiffel project (project classes, system interface classes, configuration files, C2Eif helper classes and libraries).

3.4 Evaluation and Discussion

This section evaluates the translation $\mathcal{T}$ and its implementation in C2Eif with 13 programs and 4 testsuites.

3.4.1 Correct Behavior

Table 3.1 shows the results of translating 13 open-source C programs and a testsuite into Eiffel with C2Eif, running 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 7.1.8. For each application, library, and testsuite Table 3.1

	Size (LOCS) CIL	Size (LOCS) Eiffel	#Eiffel classes	Time (s)	Binary size (MB)
`hello world`	8	15	1	1	1.3
`micro httpd`	565	1,934	16	1	1.5
`xeyes`	1,463	10,661	78	1	1.8
`less`	16,955	22,545	75	5	2.6
`wget`	46,528	57,702	183	25	4.5
`links`	70,980	100,815	211	33	13.9
`vim`	276,635	395,094	663	144	24.2
`libSDL_mixer`	7,812	11,553	47	3	–
`libmongoDB`	7,966	10,341	43	3	–
`libpcre`	18,220	24,885	38	14	–
`libcurl`	37,836	65,070	289	18	–
`libgmp`	61,442	79,971	370	21	–
`libgsl`	238,080	344,115	978	85	–
`gcc` (torture)	147,545	256,246	2,569	79	1,576
Total	932,035	1,380,947	5,561	433	1,626

Table 3.1: Translation of 13 open-source programs and a testsuite.

reports: (1) the size (in lines of code) of the CIL version of the C code and of the translated Eiffel code; (2) the number of Eiffel classes created; (3) the time (in seconds) spent by C2Eif to perform the source-to-source translation (not including compilation from Eiffel source to binary); (4) the size of the binaries (in MBytes) generated by EiffelStudio.[4]

The 13 programs include 7 applications and 6 libraries; most of them are widely-used in Linux and other "*nix" distributions. `hello world` is the only toy application, which is however useful as baseline of translating from C to Eiffel with C2Eif. 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: `libSDL_mixer` 1.2, an audio playback library; `libmongoDB` 0.6, a library to access MongoDB databases; `libpcre` 8.31, for regular expressions; `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. The `gcc` "torture tests" are short but semantically complex pieces of C code, used as regression tests for the GCC compiler.

We ran extensive trials on the translated programs to verify that they behave as in their original C version, hence validating the correctness of the translation $\mathcal{T}$ and its implementation in C2Eif. In addition to informal us-

[4]We do not give a binary size for libraries, because EiffelStudio cannot compile them without a client.

	Execution time (s)			Max % CPU			Max MB RAM		
	C	T	E	C	T	E	C	T	E
`hello world`	0	0	0	0	30	30	1.3	5.5	5.3
`micro httpd`	5	37	46	99	99	99	2.3	7.8	5.6
`wget`	16	16	–	22	22	–	4.4	69	–
`libcurl`	199	212	–	–	–	–	–	–	–
`libgmp`	44	728	–	–	–	–	–	–	–
`libgsl`	25	1501	–	–	–	–	–	–	–
`gcc` (torture)	0	5	–	–	–	–	–	–	–

Table 3.2: Performance comparison for 3 applications and 4 testsuites.

age, we performed systematic performance tests for some of the applications (described below), and ran standard testsuites on the three biggest libraries. `libcurl` comes with a client application and a testsuite of 583 tests defined in XML and executed by a Perl script calling the client; `libgmp` and `libgsl` respectively include testsuites of 145 and 46 tests, consisting of client C code using the libraries. All tests execute and pass on both the C and the translated Eiffel versions of the libraries, with the same logged output. For `libcurl`, C2Eif translated the library and the client application. For `libgmp` and `libgsl`, it translated the test cases as well as the libraries.

The `gcc` torture testsuite includes 1116 tests; the GCC version we used fails 4 of them; CIL (which depends on GCC) fails another 110 tests among the 1112 that GCC passes; finally, C2Eif (which depends on CIL) passes 989 (nearly 99%) and fails 13 of the 1002 tests passed by CIL. Given the challenging nature of the torture testsuite, this result is strong evidence that C2Eif handles the complete C language used in practice, and produces correct translations.

The 13 torture tests failing after translation to Eiffel target the following unsupported features. One test reads an **int** from a *va_list* (variadic function list of arguments) that actually stores a **struct** whose first field is a **double**; the Eiffel type-system does not allow this, and inspection suggests that it is probably a copy-paste error rather than a feature. Two tests exercise GCC-specific optimizations, which are immaterial after translation to Eiffel. Six tests target exotic GCC built-in functions, such as *builtin_frame_address*; one test performs explicit function alignment; and three rely on special bitfield operations.

3.4.2 Performance

Table 3.2 shows the result of trials that analyze the performance of six of the programs, plus the GCC torture testsuite, running on the same system as in Table 3.1. For each program or testsuite, Table 3.2 reports the execution time

(in seconds), the maximum percentage of CPU and the maximum amount of RAM (in MBytes) used while running. The table compares the performance of the original C versions (column C) against the Eiffel translations with C2Eif (column T), and, for the simpler examples, against manually written Eiffel implementations (column E) that transliterate the original C implementations using the closest Eiffel constructs (for example, *putchar* becomes *Io.put_character*) with as little changes as possible to the code structure. Maximum CPU and RAM usages are immaterial for the libraries and for the GCC testsuite, because their execution consisted of a large number of separate calls.

The performance of `hello world` demonstrates the base overhead, in terms of CPU and memory usage, of the default Eiffel runtime (objects, automatic memory management, and contract checking—which can however be disabled for applications where sheer performance is more important than having additional checks).

The test with `micro httpd` consisted in serving the local download of a 174 MB file (the Eclipse IDE); this test boils down to a very long sequence (approximately 200 million iterations) of inputting a character from file and outputting it to standard output. The translated Eiffel version incurs a significant overhead with respect to the original C version, but it is faster than the manually written Eiffel implementation. This might be due to feature lookups in Eiffel or to the less optimized implementation of Eiffel's character services. As a side note, we did the same exercise of manually transliterating `micro httpd` using Java's standard libraries; this Java translation ran the download example in 170 seconds, using up to 99% of CPU and 150 MB of RAM.

The test with `wget` downloaded the same 174 MB Eclipse package over the SWITCH Swiss network backbone. The bottleneck is the network bandwidth, hence differences in performance are negligible, except for memory consumption, which is higher in Eiffel due to garbage collection (memory is deallocated only when necessary, hence the maximum memory usage is higher in operational conditions).

The test with `libcurl` consisted in running all 583 tests from the standard testsuite mentioned before. The total runtime is comparable in translated Eiffel and C.

The tests with `libgmp` and `libgsl` ran their respective standard testsuites. The overall slow-down seems significant, but a closer look shows that the large majority of tests run in comparable time in C and Eiffel: 30% of the `libgmp` tests take up over 95% of the running time; and 26% of the `libgsl` tests take up almost 99% of the time. The GCC torture tests incur only a moderate slow-down, concentrated in 3 tests that take 97% of the time. In all

these experiments, the tests responsible for the conspicuous slow-down target operations that execute slightly slower in the translated Eiffel than in the native C (e.g., accessing a **struct** field) and repeat it a huge number of times, so that the basic slow-down increases many-fold. These bottlenecks are an issue only in a small fraction of the tests and could be removed manually in the translation.

Finally, the interactive applications (`xeyes`, `less`, `links`, and `vim`) run smoothly with good responsiveness, comparable to their original implementations.

In all, the performance overhead in switching from C to Eiffel significantly varies with the program type but, even when it is noticeable, it does not preclude the usability of the translated application or library in normal conditions (as opposed to the behavior in a few specific test cases).

3.4.3 Readability and Usability

This section shows more examples of code generated by C2Eif to demonstrate its readability and usability.

The first example is the translation of function *send_error* in `micro httpd`, which prints an error report formatted in HTML. We give the Eiffel translation below; readers familiar with C can easily figure out the behavior of the original C implementation, and Eiffel programmers will appreciate the usage of standard library services (e.g., *put_string*) to print strings and flush the output buffer.

```
send_error (
        a_status: INTEGER_32;
        a_title: CE_POINTER [INTEGER_8];
        a_extra_header: CE_POINTER [INTEGER_8];
        a_text: CE_POINTER [INTEGER_8]
    )
  do
    send_headers (a_status, a_title, a_extra_header, ce_string ("text/html"), -1,
        -1)
    Io.put_string ("<html><head><title>" + a_status.out + " " + eif_string (
        a_title) + "</title></head>%N<body bgcolor=%"#cc9999%"><h4>" +
        a_status.out + " " + eif_string (a_title) + "</h4>%N")
    Io.put_string (eif_string (a_text) + "%N")
    Io.put_string ("<hr>%N<address><a href=%"http://www.acme.com/software
        /micro_httpd/%">micro_httpd</a></address>%N</body></html>%N
        ")
    Io.standard_default.flush
    exit (1)
  end
```

The second example demonstrates the usability of the translated `libcurl`. Consider a simple client class *PRINT_SOURCE_SEHOME*, which prints the source code of the home page at `http://se.inf.ethz.ch`. This is an implementation using `libcurl` translated with C2Eif:

```
-- Using libcurl translated with C2Eif:
class
    PRINT_SOURCE_SEHOME

inherit
    LIBCURL_CONSTANTS

create
    make

feature

    make
        local
            l_curl_easy: P_EASY_STATIC
            l_curl_handle: CE_POINTER [ANY]
            l_result: NATURAL_32
        do
            create l_curl_easy
            l_curl_handle := l_curl_easy.curl_easy_init
            l_curl_easy.curl_easy_setopt ([l_curl_handle, Curlopt_url,
                ce_string ("se.inf.ethz.ch")]).do_nothing
            l_result := l_curl_easy.curl_easy_perform (l_curl_handle)
            l_curl_easy.curl_easy_cleanup (l_curl_handle)
        end

end
```

EiffelStudio includes a manually encoded library that wraps `libcurl`. The following is an implementation of class *PRINT_SOURCE_SEHOME* that uses the manually wrapped library. Notice the similarity in structure and form between the two implementations; the second one is even slightly more complex, because it needs a conditional to check that the dynamically linked library is reachable at runtime, whereas the first implementation (using C2Eif) does not depend on external libraries since `libcurl` has been translated to regular Eiffel.

```
-- Using EiffelStudio's libcurl wrapper:
class
    PRINT_SOURCE_SEHOME

inherit
    CURL_OPT_CONSTANTS

create
    make

feature

    make
        local
            l_curl_easy: CURL_EASY_EXTERNALS
            l_curl_handle: POINTER
            l_result: INTEGER_32
        do
            create l_curl_easy
            if l_curl_easy.is_dynamic_library_exists then
                l_curl_handle := l_curl_easy.init
                l_curl_easy.setopt_string (l_curl_handle, Curlopt_url, "se
                    .inf.ethz.ch")
                l_result := l_curl_easy.perform (l_curl_handle)
                l_curl_easy.cleanup (l_curl_handle)
            else
                Io.error.put_string ("cURL library not found.%N")
            end
        end

end
```

3.4.4 Safety and Debuggability

What are the benefits of automatically porting C code to Eiffel? One obvious advantage is the *reusability* of the huge C code base. This section demonstrates that the higher-level features of Eiffel can bring other benefits, in terms of improved *safety* and easier *debugging* of applications automatically generated using C2Eif.

Uncontrolled format string is a well-known vulnerability [6] of C's

printf library function, which permits malicious clients to access data in the stack by supplying special format strings. Consider for example the C program:

```
int main (int argc, char * argv[]) {
    char *secret = "This is secret!";
    if (argc > 1) printf(argv[1]);
    return 0;
}
```

If we call it with `./example "{Stack: %x%x%x%x%x%x} -> %s"`, we get the output `{stack: 0b7[...]469} -> This is secret!`, which reveals the local string *secret*. The safe way to achieve the intended behavior is *printf*(`"%s"`, *argv*[1]) instead of *printf*(*argv*[1]), so that the input string is interpreted literally.

What is the behavior of code vulnerable to uncontrolled format strings, when translated to Eiffel with C2Eif? In simple *printf* uses with just one argument as in the example, the translation replaces calls to *printf* with calls to Eiffel's *Io.put_string*, which prints strings verbatim without interpreting them; therefore, the translated code is not vulnerable in these cases. The replacement was possible in 65% of all the *printf* calls in the programs of Table 3.1. C2Eif translates more complex uses of *printf* (for example, with more than one argument and no literal format string such as *printf* (*argv*[1], *argv*[2])) into wrapped calls to the external *printf* function, hence the vulnerability still exists. However, it is less extensive or more difficult to exploit in Eiffel: primitive types (such as numeric types) are stored on the stack in Eiffel as they are in C, but Eiffel's bulkier runtime typically stores them farther up the stack, hence longer and more complex format strings must be supplied to reach the stack data (for instance, a variation of the example with *secret* requires 386 `%x` in the format string to reach local variables). On the other hand, non-primitive types (such as strings and **struct**s) are wrapped by Eiffel classes in C2Eif, which are stored in the heap, hence unreachable directly by reaching stack data. In these cases, the vulnerability vanishes in the Eiffel translation.

Debugging format strings. C2Eif also parses literal format strings passed to *printf* and detects type mismatches between format specifiers and actual arguments. This analysis, necessary when moving from C to a language with a stronger type system, helps debug incorrect and potentially unsafe uses of format strings. Indeed, a mismatch detected while running the 145 `libgmp` tests revealed a real error in the library's implementation of macro *TESTS_REPS*:

```
char *envval, *end; /* ... */
long repfactor = strtol(envval, &end, 0);
if(*end || repfactor ≤ 0) fprintf (stderr, "Invalid repfactor: %s.\n", repfactor);
```

String *envval* should have been passed to *fprintf* instead of **long** *repfactor*. GCC with standard compilation options does not detect this error, which may produce garbage or even crash the program at runtime. Interestingly, version 5.0.2 of `libgmp` patches the code in the wrong way, changing the format specifier *%s* into *%ld*. This is still incorrect because when *envval* does not encode a valid "repfactor", the outcome of the conversion into **long** is unpredictable. Finally, notice that C2Eif may also report false positives, such as **long** *v* = **"Hello!"**; *printf*(**"%s"**, *v*) which is acceptable (though probably not commendable) usage.

Out-of-bound error detection. C arrays translate to instances of class *CE_ARRAY* (see Section 3.3.1), which includes contracts that signal out-of-bound accesses to the array content. Therefore, out-of-bound errors are much easier to detect in Eiffel applications using components translated with C2Eif. Simply by translating and running the `libgmp` testsuite, we found an off-by-one error causing out-of-bound access (our patch is included in the latest library version); the error does not manifest itself when running the original C version. More generally, contracts help detect the precise location of array access errors. Consider, for example:

```
/* 1 */ int * buf = (int *) malloc(sizeof (long long int) * 10);
/* 2 */ buf = buf - 10;
/* 3 */ buf = buf + 29;
/* 4 */ *buf = 'a'; buf++;
/* 5 */ *buf = 'b';
```

buf is an array that stores 20 elements of type **int** (which has half the size of **long long int**). The error is on line 5, when *buf* points to position 20, out of the array bounds; line 2 is instead OK: *buf* points to an invalid location, but it is not written to. This program executes without errors in C; the Eiffel translation, instead, stops exactly at line 5 and signals an out-of-bound access to *buf*.

Array bound checking may be disabled, which is necessary in borderline situations where out-of-bound accesses do not crash because they assume a precise memory layout. For example, `links` and `vim` use statements of the form *block* $*p$ = (*block* $*$)*malloc*(**sizeof**(**struct** *block*)+ *len*), with *len* >0, to allocate **struct** datatypes of the form **struct** *block* { /*... */**char** *b*[1]; }. In this case, *p* points to a **struct** with room for 1 + *len* characters in *p*→*b*; the instruction *p*→*b*[*len*]='*i*' is then executed correctly in C, but the Eiffel translation assumes *p*→*b* has the statically declared size 1, hence it stops with an error. Another borderline situation is with multi-dimensional arrays, such as **double** *a*[2][3]. An iteration over *a*'s six elements with **double** $*p$ = &*a*[0][0] translated to Eiffel fails to go past the third element, because it sees *a*[0][0] as the first element of an array of length 3 (followed by another array of the same length). A

simple cast **double** $*$ p = (**double**$*$)a achieves the desired result without fooling the compiler, hence it works without errors also in translated code. These situations are examples of unsafe programming more often than not.

More safety in Eiffel. Our experiments found another bug in `libgmp`, where function *gmp_sprintf_final* had three formal input arguments, but was only called with one actual through a function pointer. Inspection suggests it is a copy-paste error of the other function *gmp_sprintf_reps*. The Eiffel version found the mismatch when calling the routine and reported a contract violation. Easily finding such bugs demonstrates the positive side-effects of translating widely-used C programs into a tighter, higher-level language.

3.4.5 Maintainability

A tool such as C2Eif, which provides automatic translation between languages, is applicable in different contexts within general software maintenance and evolution processes. This section discusses some of these applications and how suitable C2Eif can be for each of them.

Reuse in clients. The first, most natural application is using C2Eif to automatically *reuse* large C code-bases in Eiffel. This is not merely a possibility, but something extremely valuable for Eiffel, whose user community is quite small compared to those of other mainstream languages such as C, Java, or C++. Since we released C2Eif to the public as open-source, we have been receiving several requests from the community to produce Eiffel versions of C libraries whose functionalities are sorely missed in Eiffel, and whose native implementation would require a substantial effort to get to software of quality comparable to the widely tested and used C implementations. This was the case, in particular, of `libSDL_mixer`, `libmongoDB`, and `libpcre`, whose automatic translations created using C2Eif were requested from the community and are now being used in Eiffel applications. We are also aware of some Eiffel programmers directly trying to use C2Eif to translate useful C libraries and deploy them in their own software.

This form of reuse mainly entails writing Eiffel client code that accesses translated C components. Supporting it requires tools that handle the full C language as used in practice, and that produce translated APIs understandable and usable with a programming style sufficiently close to what is the norm in Eiffel, without requiring in-depth understanding of the C conventions.

When a C library undergoes maintenance, the introduced changes have the same impact on the C and on the Eiffel clients of the library. In particular, if the changes to the library do not break client compatibility (that is, the API does not change), one should simply run C2Eif again on the new library

version and replace it in the Eiffel projects that depends on it. If the API changes, clients may have to change too, independent of the language they are written in.

Evolution of translated libraries. Once a program is translated from C to Eiffel, one may decide it has become part of the Eiffel ecosystem, and hence it will undergo maintenance and evolution as any other piece of Eiffel code. In this scenario, C2Eif provides an immediately applicable solution to port C code to Eiffel, whereas the ensuing maintenance effort is distributed over an entire lifecycle and devoted to improve the automatic translation (for example, removing the application-dependent performance bottlenecks highlighted in Section 3.4.2) and completely conforming it to the Eiffel style.

Such maintenance of translations is the easier the closer the generated code follows Eiffel conventions and, more generally, the object-oriented paradigm. Providing a convincing empirical evaluation of the readability and maintainability of the code generated by C2Eif (not just from the perspective of writing client applications) is beyond the scope of the present work. Notice, however, that C2Eif already follows numerous Eiffel conventions such as for the names of classes, types, and local variables, which might look verbose to hard-core C programmers but are de rigueur in Eiffel. Follow-up work, which we discuss in Chapter 4, has targeted the object-oriented reengineering of C2Eif translations. In all, while the translations produced by C2Eif still retain some "C flavor", we believe they are overall understandable and modifiable in Eiffel with reasonable effort.

Two-way propagation of changes. One more maintenance scenario occurs if one wants to be able to independently modify a C program and its translation to Eiffel, while still being able to propagate the changes produced in each to the counterpart. For example, this scenario applies if a C library is being extended with new functionality, while its Eiffel translation produced by C2Eif undergoes refactoring to optimize it to the Eiffel environment. This scenario is the most challenging of those discussed in this section; it poses problems similar to those of merging different development branches of the same project. While merge conflicts are still a bane of collaborative development, modern version control systems (such as Git or Mercurial) have evolved to provide powerful support to ease the process of conflict reconciliation. Thus, they could be very useful also in combination with automatic translators such as C2Eif to be able to integrate changes in C with other changes in Eiffel.

3.4.6 Limitations

The only significant limitations of the translation $\mathcal{T}$ implemented in C2Eif in supporting C programs originate in the introduction of strong typing: programming practices that implicitly rely on a certain memory layout may not work in C applications translated to Eiffel. Section 3.4.4 mentioned some examples in the context of array manipulation (where, however, the checks on the Eiffel side can be disabled). Another example is a function **int** *trick* (**int** *a*, **int** *b*) that returns its second argument through a pointer to the stack, with the instructions **int** $*p = \&a$; **return** $*(p+1)$. C2Eif's translation assumes p points to a single integer cell and cannot guarantee that b is stored in the next cell.

Furthermore, the GCC torture testsuite highlighted a few exotic GCC features currently unsupported by C2Eif (Section 3.4.1), which may be handled in future work.

3.5 Related Work

We have presented a survey of the legacy code migration field; in Section 2.1 for wrapping foreign code and in Section 2.2 for translating foreign code. C2Eif is a considerable contribution to the field of automatic foreign code translation and can also be used to automatically wrap foreign code.

Automatically translating foreign code. Adding C2Eif to Table 2.2 shows the advantages over other solutions for the C programming language:

	target language	completely automatic	available	readability	external libraries	pointer arithmetic	**goto**s	inlined assembly
`Ephedra`	Java	no	no	+	no	no	no	no
`Convert2Java`	Java	no	no	+	no	no	no	no
`C2J++`	Java	no	no	+	no	no	no	no
`C2J`	Java	no	yes	−	no	yes	no	no
`C++2Java`	Java	no	yes	+	no	no	no	no
`C++2C#`	C#	no	yes	+	no	no	no	no
`C2Eif`	Eiffel	yes	yes	+	yes	yes	yes	yes

Table 3.3: Tools translating C to O-O languages.

A limitation of other solutions is that they are not automatic and hardly handle the trickier or specialized parts of the C language, which are tempting to dismiss as unimportant "corner cases", but figure prominently in real-world programs. C2Eif on the other hand is completely automatic, covers

the entire C language as used in practice and has been tested on programs of considerable size.

Automatically wrapping foreign code. As presented in Section 3.3.4, C2Eif can automatically create wrappers for external functions and variables to access pre-compiled C libraries (e.g., for I/O). C2Eif can therefore also be used to automatically create wrappers for foreign code. Instead of processing the source code of a library with C2Eif to get a translation, it is sufficient to just process the header files of that library to get a wrapper.

Safer C. Many techniques exist aimed at ameliorating the safety of existing C code; for example, detection of format string vulnerability [5], out-of-bound array accesses and other memory errors [38, 34], or type errors [33]. C2Eif has a different scope, as it offers improved safety and debuggability as *side-benefits* of automatically porting C programs to Eiffel. This shares a little similarity with Ellison and Rosu's formal executable semantics of C [9], which also finds errors in C program as a "side effect" of a rigorous translation.

3.6 Summary

This chapter presented a complete translation of C programs into Eiffel, and its implementation in a freely available tool C2Eif. Experiments showed that C2Eif correctly translates complete applications and libraries of significant size, and takes advantage of some of Eiffel's advanced features to produce safer code.

Chapter 4

Object-Oriented Reengineering

4.1 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.

This chapter presents a novel technique, and a supporting tool AutoOO, that extracts such implicit design elements from C programs and uses them to build reengineered object-oriented programs.

While AutoOO translates C to Eiffel, the very same reengineering techniques are applicable to other object-oriented programming languages.

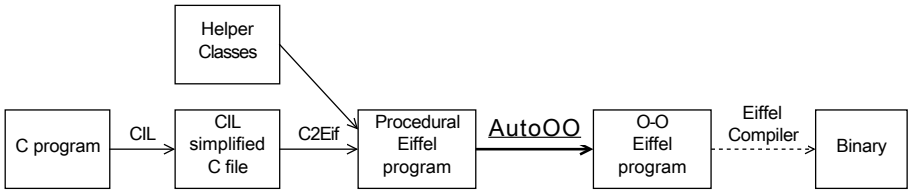

Figure 4.1: Object-oriented reengineering with C2Eif and AutoOO.

C2Eif and AutoOO. The tool AutoOO builds on top of C2Eif, discussed in Chapter 3. Figure 4.1 shows how C2Eif and AutoOO are combined in a toolchain that goes from procedural C input to object-oriented Eiffel output. The C source program is processed by CIL [32], which simplifies some C constructs (for example, there is only one type of loop in CIL); C2Eif translates the CIL output into a procedural Eiffel program; AutoOO processes the C2Eif output, introduces the transformations described in Sections 4.3 and 4.4, and outputs the reengineered object-oriented Eiffel programs that can be compiled.

In other words, C2Eif produces Eiffel code functionally equivalent to C but without any object-oriented reengineering. AutoOO implements the novel technique described in this chapter to extract high-quality object-oriented designs from procedural code; it relies on C2Eif only to produce a raw translation of C into Eiffel.

From the user perspective, AutoOO and C2Eif are the same tool since AutoOO has been added to the user inferface of C2Eif.

Outline. In the rest of the chapter, Section 4.2 defines the goals of AutoOO reengineering, how they are assessed, and the design principles followed. Section 4.3 discusses how AutoOO introduces elements of object-oriented design—in particular, how it populates classes. Section 4.4 discusses how it introduces contracts and exceptions. Section 4.5 presents the evaluation of the correctness, scalability, and performance of AutoOO based on 10 reengineered applications and libraries. Section 4.6 reviews the fundamental aspects of the object-oriented design style introduced by AutoOO and how they make for usable reengineered programs. Section 4.7 discusses the current limitations of AutoOO. Section 4.8 reviews related work and compares AutoOO against existing tools and approaches to object-oriented reengineering.

4.2 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.

While reengineering in its most general meaning—the reconstruction of "a system in a new form" [3]—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. 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: [10, 21, 51]) 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.

The rest of this section presents other specific goals of AutoOO and how we assess them.

Case studies. The evaluation of AutoOO, described in the following sections, targets 10 of the open-source programs used for C2Eif in Table 3.1, totalling 750 KLOC (see Section 3.4 for details). The 10 programs include 7 applications – `hello world`, `micro httpd`, `xeyes`, `less`, `wget`, `links`, and `vim` – and 3 libraries – `libcurl`, `libgmp`, and `libgsl`. Section 4.5 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 Au-

toOO 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 4.5 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 (attributes) and functions operating on them (routines). Section 4.3 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 routines belong to the correct class, that is are indeed routines operating on the attributes of the class.
- *Coupling and cohesion*: the coupling of a class is measured as the ratio: number of accesses to features of other classes / number of accesses to features 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.[1]
- *Information hiding* is measured as the ratio of private to public features. A high ratio indicates that classes make good usage of information hiding for encapsulation.
- *Instance vs. class features*: the ratio of instance to class features (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 attributes and routines.
- *Inheritance*: we manually inspected all uses of inheritance introduced by AutoOO and we determined how many correctly define substitutable heir classes.

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.

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

Reengineering step	# bundle routines	# datatype routines	# bundle attributes	# datatype attributes	% sound datatype routines	average coupling	overall hiding	instance/class features	# 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 4.1: Object-oriented design metrics after each reengineering step applied to the 10 case study programs.

- *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 4.4.1.

- *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 4.4.2.

4.3 Object-Oriented Design

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 routines 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 features.

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 refactoring.

Corresponding to their roles, datatype classes contain mainly "proper" *instance* features (Section 4.3.3 discusses the exceptions), whereas bundle classes contain only *class* features (also called *static*). Note that Eiffel does not offer a direct counterpart to Java's *static* mechanism; equivalent semantics for a routine can be achieved by placing it into a stateless class and using a fresh instance of that class as a target for every call (see an example in Figure 4.3). We therefore use the term "class routine" in the Eiffel context to refer to a routine of such a stateless class.

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**). This step is a part of the basic translation performed by C2Eif.

2. *Function signature* analysis refactors routines from bundle to datatype classes, moving operations closer to the data definition they work on.

3. *Call graph* analysis refactors features from bundle to datatype classes, and shuffles routines among datatype classes, moving features to classes where they are exclusively used.

4. *Inheritance* analysis creates inheritance relationships between datatype classes based on their attributes.

The following subsections 4.3.1–4.3.4 describe the steps in detail with examples.

Table 4.1 reports how the various metrics mentioned in Section 4.2 change as we apply the four steps to the 10 case study programs. For each reengineering step, Table 4.1 reports:

- The number of bundle and datatype features[2] created, partitioned in routines and attributes.

- The percentage of *sound* datatype routines.[3] A routine 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

[2] A bundle (or datatype) feature is a feature of a bundle (or datatype) class.
[3] Evaluated on all projects but `vim` and `libgsl`, as discussed in Section 4.2.

access is the read or write of an attribute, or a routine call; an access in the body of a routine m of T is *in* if it refers to a feature of T other than m; it is *out* if it refers to a feature of a class other than T. When counting accesses in a routine 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 features. For each step, Table 4.1 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* to *public* features.
- The ratio of *instance* to *class* features.
- The number of classes defined using *inheritance*.

The rest of this section will discuss the figures shown in Table 4.1 to demonstrate how each reengineering step improves these object-oriented design metrics.

4.3.1 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 attributes. 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.

For example, when processing the C source file in Figure 4.2, the first step generates the datatype class *PERSON*:

class
PERSON

feature
age: *INTEGER*
sex: *BOOLEAN*
end

```
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);
}
```

Figure 4.2: C source file `PersonHandler.c`.

and the bundle class *PERSON_HANDLER*:

```
class
    PERSON_HANDLER

feature
    majority_age: STATIC [INTEGER]
        once
            Result.item := 18
        end

    set_age (p: POINTER [PERSON]; new_age: INTEGER)
        local
            return: BOOLEAN
        do
            if new_age <= 0 then
                return := True
            end
            if not return then
                p.item.age := new_age
            end
        end
```

```
    overage (age: INTEGER): BOOLEAN
        do
            Result := (age > majority_age.item)
        end

    is_adult (p: POINTER [PERSON]): BOOLEAN
        do
            Result := overage (p.item.age)
        end

end
```

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 no hiding and a low instance/class feature ratio (the only instance features are the datatype attributes).

The overall coupling (first row in Table 4.1) is also quite high after step 1. This does not come as a surprise: because all routines 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.

4.3.2 Function Signature Analysis

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

Consider a routine 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 routine 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 **Current** (called **this** in Java). Accordingly, m's body may have to adjust other references to features 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 4.2, the second reengineering step determines that *set_ age* and *is_ adult* can be refactored: argument p is data-bound and globally used in both routines (with the first instruction in *set_ age* being an argument handling path). Hence, the two routines are moved from class *PERSON_ HANDLER* to class *PERSON* which becomes:

```
class
    PERSON

feature
    age: INTEGER

    sex: BOOLEAN

    set_ age (new_ age: INTEGER)
        local
            return: BOOLEAN
        do
            if new_ age ≤ 0 then
                return := True
            end
            if not return then
                age := new_ age
            end
        end

    is_ adult: BOOLEAN
        do
            Result := (create {PERSON_ HANDLER}).overage (age)
        end
end
```

Function signature analysis introduces fundamental elements of object-oriented design. As reported in Table 4.1, manual inspection reveals that 94% of the routines moved to datatype classes are indeed operations on that type; this means that 97% of the features of datatype classes (attributes plus sound routines) are refactored correctly. Remember that our definition of soundness refers to design, not to correct behavior: even the 6% = 100% − 94% "unsound" routines 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 routines that access attributes of datatype classes are now located inside those classes. This dominates over the increase in out accesses to bundle features from within the routines 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 routines have become instance routines, 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 routines? For example, if the code in Figure 4.2 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 are 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 routines 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 4.2 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

Void (known as **null** in C and Java); therefore, it cannot become an instance routine which always has an implicit non-**Void** target **Current**.

As an interesting observation about the application of reengineering to the programs of Table 4.2, we found that 40% of the routines moved to datatype classes in step 2 have a name that includes the datatype class name as prefix. For example, the routines 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 4.6).

4.3.3 Call Graph Analysis

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

Consider a feature n of any class N that is accessed (read, written, or called) *only* in a datatype class T. For each such feature n, the third reengineering step moves n into the datatype class T. If n is an instance routine or a class routine, it becomes an instance routine; if it is a class attribute, it remains a class attribute to preserve the original semantics of *static* attributes corresponding to global C variables (this is the only case where we add class features to datatype classes). Features 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 feature out of a class changes the global call graph, AutoOO performs the third reengineering step iteratively: it starts with the feature n with the largest number of accesses, and updates the call graph after every refactoring move, recalculating the set of candidate features for the next move.

Continuing the example of Figure 4.2, assume that routine *overage* of bundle class *PERSON_HANDLER* is only called by *is_adult* in datatype class *PERSON*, and that attribute *majority_age* is instead read also by other modules. Then, AutoOO moves *overage* to *PERSON* where it becomes non-*static* and *private*:

```
overage (age: INTEGER): BOOLEAN
    do
        Result := (age >(create {PERSON_HANDLER}).
            majority_age.item)
    end
```

The attribute *majority_ age* stays unchanged in class *PERSON_ HANDLER*.

As reported in Table 4.1, call graph analysis refines the object-oriented design and introduces hiding when possible, that is for 12% of the features. Even if there are 2,290 private features, 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 routines to the class where they are used. The percentage of sound refactored routines increases to 96%; overall, 98% of the datatype features are refactored correctly. Step 3 also makes instance features the majority (53% of all features, or 1.12 instance feature per class feature).

Under the conservative approach taken by AutoOO, which creates functionally equivalent code, we judge that the values of hiding, coupling, and instance/class features 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 features in classes of the reengineered application. The example in Section 4.6 reinforces these conclusions from a user's perspective.

4.3.4 Inheritance Analysis

The fourth reengineering step introduces inheritance in order to make existing subtyping relationships between datatype classes explicit. The approach is similar to what has been suggested in [24]; 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; ...; t_m a_m; };
struct s { u_1 b_1; u_2 b_2; ...; 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 $n > m$[4] and, for all $1 \leq i \leq m$, the types t_i and u_i are equivalent. 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 **rename** clause to rename

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

attributes with different names, s becomes:

class
S

inherit
R
rename
a_1 **as** b_1,
a_2 **as** b_2,
...
a_m **as** b_n
end

feature

b_{m+1} : u_{m+1}

...

b_n : u_n

−− Rest of the class unchanged.

end

Notice that AutoOO bases inheritance analysis on type information only, not on attribute *names*. Therefore, it requires renaming of attributes 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 4.2, 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

inherit
PERSON

feature

gpa: *INTEGER*

−− ...

end

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 4.1 do not change after inheritance analysis, assuming we count attributes 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:

```
struct element_list {
    struct element *first;
};
```

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

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

4.4 Contracts and Exceptions

AutoOO introduces contracts and exceptions to improve the readability of the classes generated in the reengineering. Section 4.4.1 explains how AutoOO builds contracts from compiler-specific function annotations and from simple implicit properties of pointers found by static analysis. Section 4.4.2 discusses how exceptions can capture the semantics of *longjmp*.

4.4.1 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 routines, namely preconditions and postconditions. A routine's precondition (introduced by **require**) is a predicate that must hold whenever the routine is called; it is the caller's responsibility to establish the routine's precondition before calling it. A routine's postcondition (introduced by **ensure**) is a predicate that must hold whenever the routine terminates; it is the routine'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 4.2.

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-**Void**; the latter marks functions that never return (for example, the system function *exit*). For each routine $m\ (t_1\ p_1, \ldots, t_m\ p_m)$ corresponding to a C function with attribute ***nonnull*** $(i_1, \ldots, i_n)$, with $n \geq 0$ and $1 \leq i_1, \ldots, i_n \leq m$ denoting arguments of m by position, AutoOO adds to m the precondition

require
  $p_{i_1} \neq$**Void**
  $p_{i_2} \neq$**Void**
  ...
  $p_{i_n} \neq$**Void**

that the arguments $p_{i_1}, \ldots, p_{i_n}$ be non-**Void**. For each routine m corresponding to a C function with attribute ***noreturn***, AutoOO adds to m the postcondition **ensure False** that would be violated if m ever terminates.

Extending the example of Figure 4.2, 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 **Current**):

```
kill (q: POINTER [PERSON])
    require
        q ≠Void
    ensure
        False
    end
```

GCC function attributes determined 266 precondition and 13 postcondition clauses in the programs of Table 4.2.

Globally used pointers Section 4.3.2 defined the notion of *globally used* argument: an argument that is accessed (read or written) at least once along every path in a routine's body. Based on the same notions, for each pointer argument p of a routine m that is globally used in m on *all* paths (including argument-handling paths), AutoOO adds to m the precondition **require** $p \neq$**Void** that p be non-**Void**. The precondition does not change the behavior of the routine: if m were called with $p =$ **Void**, m would eventually crash in every execution when accessing a **Void** reference, and hence $p \neq$**Void** 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 4.2.

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 4.2 is called with a non-positive *new_age*, it returns without changing p's age attribute, 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

require *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 4.4.1) 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-**Void**. 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 **Void**. Correspondingly, AutoOO builds the contracts for these functions a bit differently:

require
  n >0 **implies** (*dest* ≠**Void and** *src* ≠**Void**)

that is *dest* and *src* must be non-**Void** 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.

4.4.2 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 routine 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.

	Size (LOCS)			Translation	Binary
	Procedural (C)	O-O (Eiffel)	# classes	(s)	size (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 4.2: Reengineering of 10 open-source programs.

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). C2Eif detects usages of these library functions and renders them using exceptions. The technical details of the translation can be found in Section 3.3.3. From the point of view of the reengineering, however, the translation expresses the complex semantics of *longjmp* naturally through the familiar exception handling mechanism.

6 usages of *longjmp* have been found in the programs of Table 4.2, which have been 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.

4.5 Correctness, Scalability, and Performance

In addition to the metrics of object-oriented design displayed in Table 4.1 and discussed in the previous sections, we evaluated the *behavior* of the reengineering produced by AutoOO on the 10 programs in Table 4.2; the same programs as already used for the evaluation of C2Eif (see Table 3.1). All the experiments ran on the same system.

The reengineering of each program proceeds as previously shown in Figure 4.1, 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 4.2 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[5].

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 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 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 4.2) 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 4.2. 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 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.

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

4.6 Discussion: Created Object-Oriented Style

All object-oriented designs produced by AutoOO deploy a collection of classes partitioned into bundle and datatype classes, as we explained in Section 4.3. 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.

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.[6] 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 4.3 on the left. A client using the MongoDB library translated and reengineered by AutoOO would instead use the syntax shown in Figure 4.3 on the right.

On the one hand, the two programs in Figure 4.3 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 6 and 11); and function calls become instance routine calls (lines 7, 12, 16, and 22). Routine 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 18, which remains a *static* routine call with identical signature. AutoOO did not turn it into an instance routine 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;
    /* ... */
}
```

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

```
1                                          conn: MONGO
2                                          status: INTEGER
3                                          cursor: MONGO_CURSOR
4
5  // connect to database                  -- connect to database
6  mongo conn[1];                          create conn
7  int status = mongo_connect(conn,        status := conn.connect (
8                "127.0.0.1", 27017);                "127.0.0.1", 27017)
9
10 // iterate over database content        -- iterate over database content
11 mongo_cursor cursor[1];                 create cursor
12 mongo_cursor_init(cursor, conn,         cursor.init (conn.address,
13                "tutorial.people");                 "tutorial.people")
14 while(mongo_cursor_next(cursor)         from until cursor.next ≠ Mongo_ok
15       == MONGO_OK) {                    loop
16    bson_print(&cursor→current);            cursor.current.print
17 }                                       end
18 mongo_cursor_destroy(                   (create {MONGO_S}).mongo_cursor_destroy (
19                &cursor);                         cursor.address)
20
21 // disconnect from database             -- disconnect from database
22 mongo_destroy(conn);                    conn.destroy
```

Figure 4.3: A MongoDB client application written in C (left) and the same application written for the AutoOO translation of MongoDB (right).

As discussed in Section 4.3.2, AutoOO does not reengineer such functions because the target of an object-oriented call is not allowed to be **Void**. 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.

4.7 Limitations

By and large, the evaluation with the programs of Table 4.2 demonstrates that AutoOO is a scalable technique applicable to programs of considerable size and producing good-quality object-oriented designs automatically.

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 4.3.2 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%.

	source–target	tool support	completely automatic	full language	evaluated	class identification	instance routines	inheritance
Gall [14]	methodology	no	no	–	yes	yes	?	no
Jacobson [18]	methodology	no	no	–	yes	yes	no	–
Livadas [22]	C–C++	yes	no	no	no	yes	yes	no
Kontogiannis [20]	C–C++	yes	no	?	10KL	yes	?	yes
Frakes [13]	C–C++	yes	no	no	2KL	yes	?	no
Fanta [11]	C++–C++	yes	no	no	120KL	yes	?	no
Newcomb [35]	Cobol–OOSM	yes	yes	no	168KL	yes	?	no
Mossienko [29]	Cobol–Java	yes	no	no	25KL	yes	no	no
Sneed [45]	Cobol–Java	yes	yes	no	200KL	yes	?	no
Sneed [42]	PL/I–Java	yes	yes	no	10KL	yes	?	no
AutoOO	C–Eiffel	YES	yes	yes	750KL	yes	yes	yes

Table 4.3: Comparison of approaches to O-O reengineering.

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. Another limitation 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 4.5) 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 [9].

4.8 Related Work

Reengineering [3] is a common practice—and an expensive activity [44]—in professional software development. Given the wide adoption of languages with object-oriented features, object-oriented reengineering is frequently necessary.

Section 2.3 presented a survey of the existing approaches in the field; Table 4.3 compares those approaches with AutoOO. A direct detailed comparison 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. Also, since AutoOO is currently the only publicly available tool (denoted by the small caps YES), we couldn't try the other approaches our-

selves. This is the reason for the presence of "?" in the table, especially in the column "instance routines", corresponding to cases where we could not figure out the details of how routines are refactored. In light of the evidence collected (or lack thereof), it is fair to say that identifying instance routines 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.

4.9 Summary and Conclusions

In this chapter we presented a new, completely automatic approach to object-oriented reengineering of C programs. The supporting tool, 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 attributes and routines 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.

Software reengineering techniques come in many flavours and serve a variety of purposes, such as improving design, maintainability, or performance. All those techniques – and in particular fully automatic ones – involve a trade-off between the extent of the reengineering and the correctness of the resulting code. We took a conservative approach, requiring that the reengineered code be functionally equivalent to the original, which makes our technique applicable to realistic software of significant size, at the cost of limiting the extent of possible code improvements.

Chapter 5

Java to Eiffel Translation

In addition to the translation of procedural C source code into object-oriented Eiffel source code, we have also investigated the related problem of automatically translating source code between different modern object-oriented environments, in particular from Java to Eiffel. The results of this effort are presented in this chapter.

5.1 Introduction

Reusability is an important software engineering concept actively advocated for the last forty years. While reusability has been addressed for systems implemented using the same programming language, it does not usually handle interoperability with different programming languages. This chapter presents a solution for the reuse of Java code within Eiffel programs based on a source-to-source translation from Java to Eiffel.

The translation is implemented in the freely available tool J2Eif [17]; it provides Eiffel replacements for the components of the Java runtime environment, including Java Native Interface services and reflection mechanisms. Since Eiffel compiles to native code, a valuable by-product of J2Eif is the possibility of compiling Java applications to native code.

While Eiffel and Java are both object-oriented languages, the translation of one into the other is tricky because superficially similar constructs, such as those for exception handling, often have very different semantics. In fact, correctness is arguably the main challenge of source-to-source translation: Section 5.3 formalizes the most delicate aspects of the translation to describe how they have been tackled and to give confidence in the correctness of the translation.

Our experiments demonstrate the practical usability of the translation scheme and its implementation, and record the performance slow-down compared to custom-made Eiffel applications: automatic translations of *java.util* data structures, *java.io* services, and SWT applications can be re-used as Eiffel programs, with the same functionalities as their original Java implementations.

Section 5.2 gives an overview of the architecture of J2Eif; Section 5.3 describes the translation in detail; Section 5.4 evaluates the implementation with four experiments and points out its major limitations; Section 5.5 discusses related work.

5.2 Design Principles

J2Eif [17] is a stand-alone compiler with graphical user interface that translates Java programs to Eiffel programs. The translation is a complete Eiffel program which replicates the functionalities of the Java source program by including replacements of the Java runtime environment (most notably, the Java Native Interface and reflection mechanisms). J2Eif is implemented in Java.

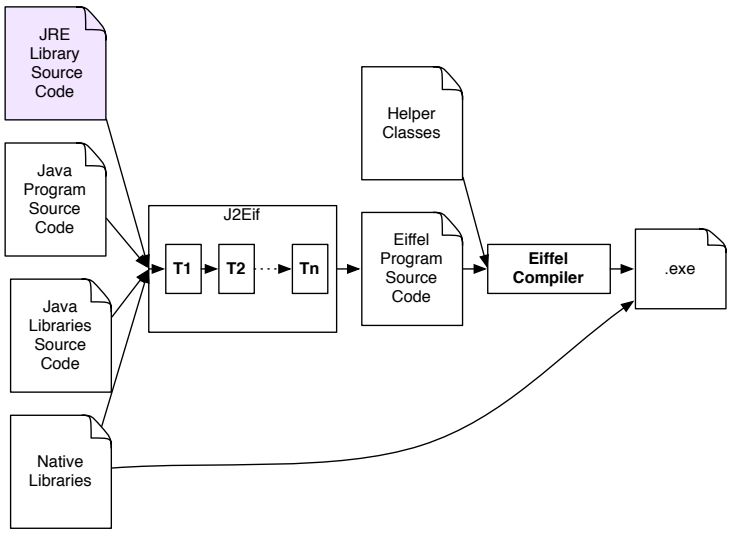

Figure 5.1: High-level view of J2Eif.

High-level view.

Figure 5.1 shows the high-level usage of J2Eif. To translate a Java program, the user provides the source code of the program, its Java dependencies, as well as any external native libraries referenced by the program. J2Eif produces Eiffel source code that can be compiled by an Eiffel compiler such as EiffelStudio. Native libraries called by native methods in Java are then directly called from Eiffel. While J2Eif can compile the complete code of the Java Runtime Environment (JRE) library source, it comes with a precompiled version which drastically reduces the overall compilation time.

Translation.

J2Eif implements a mapping $\mathcal{T}$: Java $\rightarrow$ Eiffel of Java code into Eiffel code. Both languages follow the object-oriented paradigm and hence share several notions such as objects, classes, methods, and exceptions. Nonetheless, the precise semantics of each notion often differs between the two languages. Section 5.3 describes all the aspects taken into account by the translation and focuses on its particularly delicate features by formalizing them.

J2Eif implements the translation $\mathcal{T}$ as a series $T_1, \ldots, T_n$ of successive incremental transformations on the Abstract Syntax Tree. Every transformation T_i takes care of exactly one language construct that needs adaptation and produces a program in an intermediate language L_i which is a mixture of Java and Eiffel constructs: the code progressively *morphs* from Java to Eiffel code.

$$\mathcal{T} \equiv\ T_n \circ \cdots \circ T_1, \text{ where } \begin{cases} T_1 : & \text{Java} & \rightarrow & L_1 \\ T_2 : & L_1 & \rightarrow & L_2 \\ & \cdots & & \\ T_n : & L_{n-1} & \rightarrow & \text{Eiffel} \end{cases}$$

The current implementation uses 35 such transformations (i.e., $n = 35$). Combining small transformations has some advantages: several of the individual transformations are straightforward to implement and all are simple to maintain; it facilitates reuse when building other translations (for example into a language other than Eiffel); the intermediate programs generated are readable and easily reviewable by programmers familiar with Java and Eiffel.

5.3 Translating Java to Eiffel

This section describes the salient features of the translation $\mathcal{T}$ from Java to Eiffel, grouped by topic. Eiffel and Java often use different names for

Java	Eiffel	Java	Eiffel
class	class	member	feature
abstract/interface	deferred	field	attribute
concrete	effective	method	routine
exception	exception	constructor	creation procedure

Table 5.1: Object-oriented terminology in Java and Eiffel.

comparable object-oriented concepts; to avoid ambiguities, the present chapter mentions both terms, whenever possible without affecting readability, and uses only the appropriate one when discussing language-specific aspects. Table 5.1 lists the Java and Eiffel names of fundamental object-oriented concepts.

5.3.1 Language Features

We formalize some components of $\mathcal{T}$ by breaking it down into simpler functions denoted by ∇; these functions are a convenient way to formalize $\mathcal{T}$ and, in general, different than the transformations T_i discussed in Section 5.2; the end of the present section sketches an example of differences between ∇'s and T_i's. The following presentation ignores the renaming scheme, discussed separately (Section 5.3.4), and occasionally overlooks inessential syntactic details. The syntax of Eiffel's exception handling adheres to the working draft 20.1 of the ECMA Standard 367; adapting it to work with the syntax currently supported is trivial.

Classes and interfaces.

A Java program is defined by a collection of classes and interfaces; the function ∇_C maps a single Java class or interface into an Eiffel class or deferred (abstract) class.

$\mathcal{T}(C_1, ..., C_n)$ $= \nabla_C(C_1), \ldots, \nabla_C(C_n)$
∇_C(**class** *name* extend {body}) $=$ **class** *name* ∇_I(extend) ∇_B(body) **end**
∇_D(**interface** *name* extend {body}) $=$ **deferred class** *name* ∇_I(extend) ∇_{iB}(body) **end**
where *name* is a class name; *extend* is a Java inheritance clause; and *body* a Java class body.

∇_I translates Java inheritance clauses (**extends** and **implements**) into Eiffel **inherit** clauses. The translation relies on two helper classes:

- *JAVA_PARENT* is ancestor to every translated class, to which it provides helper routines for various services such as access to the native interface, exceptions, integer arithmetic (integer division, modulo, and shifting

have different semantics in Java and Eiffel), strings. The rest of this section describes some of these services in more detail.

- *JAVA_INTERFACE_PARENT* is ancestor to every translated interface.

Java generic classes and interfaces may have complex constraints which cannot be translated directly into Eiffel constraints on generics. $\mathcal{T}$ handles usages of genericity with the same approach used by the Java compiler: it erases the generic constraints in the translation but enforces the intended semantics with explicit type casts added where needed.

Members (features).

∇_B and ∇_{iB} respectively translate Java class and interface bodies into Eiffel code. The basic idea is to translate Java fields and (abstract) methods respectively into Eiffel attributes and (deferred) routines. A few features of Java, however, have no clear Eiffel counterpart and require a more sophisticated approach:

- *Anonymous classes* are given an automatically generated name.
- *Arguments and attributes* in Java by default can be assigned to, unlike in Eiffel where arguments are read-only and modifying attributes requires setter methods. To handle these differences, the translation $\mathcal{T}$ introduces a helper generic class *JAVA_VARIABLE* [*G*]. Instances of this class replace Java variables; assignments to arguments and attributes in Java are translated to suitable calls to the routines in the helper class.
- *Constructor chaining* is made explicit with calls to **super**.
- *Field hiding* is rendered by the naming scheme introduced by $\mathcal{T}$ (Section 5.3.4).
- *Field initializations* and initializers are added explicitly to every constructor.
- *Inner classes* are extracted into stand-alone classes, which can access the same outer members (features) as the original inner classes.
- *JavaDoc* comments are ignored.
- *Static members.* Eiffel's *once* routines can be invoked only if they belong to effective (not deferred) classes; this falls short of Java's semantics

for static members of abstract classes. For each Java class *C*, the translation $\mathcal{T}$ introduces a class *C_STATIC* which contains all of *C*'s static members and is inherited by the translation of *C*; multiple inheritance accommodates such helper classes. *C_STATIC* is always declared as effective (not deferred), so that static members are always accessible in the translation as *once* routines.

- *Vararg arguments* are replaced by arguments of type array.
- *Visibility.* Eiffel's visibility model is different from Java's, as it requires, in particular, to list all names of classes that can access a non-public member. $\mathcal{T}$ avoids this issue by translating every member into a *public* Eiffel feature.

Instructions.

∇_M maps Java method bodies to Eiffel routine bodies. As expected, ∇_M is compositional: $\nabla_M(\text{inst}_1 \text{ ; } \text{inst}_2) = \nabla_M(\text{inst}_1) \text{ ; } \nabla_M(\text{inst}_2)$, hence it is sufficient to describe how ∇_M translates Java instructions into Eiffel. The translation of many standard instructions is straightforward; for example, the Java conditional **if** (*cond*){*doThen*} **else** {*doElse*} becomes the Eiffel conditional **if** ∇_E(*cond*) **then** ∇_M(*doThen*) **else** ∇_M(*doElse*) **end**, where ∇_E maps Java expressions to equivalent Eiffel expressions. The following presents the translation of the constructs which differ the most in the two languages.

Loops. *Loops* are not straightforward to translate because Eiffel does not provide control-flow breaking instructions (such as **break**), present in Java. Correspondingly, the translation of **while** loops relies on an auxiliary function ∇_W : JavaInstruction $\times \{\top, \bot\} \to$ EiffelInstruction which replicates the semantics in presence of **break** (with t $\in \{\top, \bot\}$):

$$
\begin{array}{ll}
\nabla_M(\textbf{while } (\text{stayIn}) \{\text{body}\}) & = \textbf{from } \textit{breakFlag} := \textbf{False} \\
 & \quad \textbf{until not } \nabla_E(\text{stayIn}) \textbf{ or } \textit{breakFlag} \\
 & \quad \textbf{loop } \nabla_W(\text{body}, \bot) \textbf{ end} \\
\nabla_W(\textbf{break}, \text{t}) & = \textit{breakFlag} := \textbf{True} \\
\nabla_W(\text{inst}_1 \text{ ; } \text{inst}_2, \text{t}) & = \begin{cases} \nabla_W(\text{inst}_1, \text{t}) \text{ ; } \nabla_W(\text{inst}_2, \top) & \text{if inst}_1 \text{ contains } \textbf{break} \\ \nabla_W(\text{inst}_1, \text{t}) \text{ ; } \nabla_W(\text{inst}_2, \text{t}) & \text{if inst}_1 \text{ doesn't contain } \textbf{break} \end{cases} \\
\nabla_W(\text{atomicInst}, \top) & = \textbf{if not } \textit{breakFlag} \textbf{ then } \nabla_M(\text{atomicInst}) \textbf{ end} \\
\nabla_W(\text{atomicInst}, \bot) & = \nabla_M(\text{atomicInst})
\end{array}
$$

The **break** instruction becomes, in Eiffel, an assignment of *True* to a fresh boolean flag *breakFlag*, specific to each loop. Every instruction within the loop body which follows a **break** is then guarded by the condition *not breakFlag* and the loop is exited when the flag is set to *True*. Other types of loops (**for**, **do**..**while**, *foreach*) and control-flow breaking instructions (**continue**, **return**) are translated similarly.

Exceptions. Both Java and Eiffel offer exceptions, but with very different semantics and usage. The major differences are:

- Exception handlers are associated with routines in Eiffel (**rescue** block) and with arbitrary, possibly nested, blocks in Java (**try**..**catch** blocks).
- The usage of control-flow breaking instructions (e.g., **break**) in Java's **try**..**finally** blocks complicates the propagation mechanism of exceptions [30].

The function ∇_M translates Java's **try**..**catch** blocks into Eiffel's agents (similar to closures, function objects, or delegates) with **rescue** blocks, so that exception handling is block-specific and can be nested in Eiffel as it is in Java:

∇_M(**try** {doTry} **catch** (t e) {doCatch}) =

```
skipFlag := False
(agent (args)
   do
     if not skipFlag then
       ∇M(doTry)
     end
   rescue
     if e.conforms_to (∇T(t)) then
       ∇M(doCatch)
       Retry := True
       skipFlag := True
     else
       Retry := False
     end
   end).call
```

∇_M(**throw** exp) = (**create** {*EXCEPTION*}).*raise*(∇_E(*exp*))

The agent's body contains the translation of Java's try block. If executing it raises an exception, the invocation of *raise* on a fresh exception object transfers control to the **rescue** block. The **rescue**'s body executes the translation of the **catch** block only if the type of the exception raised matches that declared in the **catch** (∇_T translates Java types to appropriate Eiffel types, see Section 5.3.2). Executing the **catch** block may raise another exception; then, another invocation of *raise* would transfer control to the appropriate outer **rescue** block: the propagation of exceptions works similarly in Eiffel and Java. On the contrary, the semantics of Eiffel and Java diverge when the **rescue**/**catch** block terminates without exceptions. Java's semantics prescribes that the computation continues normally, while, in Eiffel, the computation propagates the exception (if *Retry* is **False**) or transfers control back to the beginning of the **agent**'s body (if *Retry* is **True**). The translation ∇_M sets *Retry* to **False** if **catch**'s exception type is incompatible with the exception raised, thus propagating the exception. Otherwise, the **rescue** block sets *Retry* and the fresh boolean flag *skipFlag* to **True**: control is transferred back to the

agent's body, which is however just skipped because *skipFlag* = **True**, so that the computation continues normally after the **agent** without propagating any exception.

An exception raised in a **try**..**finally** block is normally propagated after executing the **finally**; the presence of control-flow breaking instructions in the **finally** block, however, cancels the propagation. For example, the code block:

```
b=2;
while(true) {
   try {
      throw new Exception();
   } finally {
      b++;
      break;
   }
}
b++;
```

terminates *normally* (without exception) with a value of 4 for the variable *b*.

The translation ∇_M renders such behaviors with a technique similar to the Java compiler: it duplicates the instructions in the **finally** block, once for normal termination and once for exceptional termination:

∇_M(**try** {doTry} **finally** {doFinally}) =

```
skipFlag := False
(agent (args)
   do
      if not skipFlag then
         ∇M(doTry)
         ∇M(doFinally)
      end
   rescue
      ∇M(doFinally)
      if breakFlag then
         Retry := True
         skipFlag := True
      end
   end).call
```

A **break** sets *breakFlag* and, at the end of the **rescue** block, *Retry* and *skipFlag*; as a result, the computation continues without exception propagation. Other control-flow breaking instructions are translated similarly.

Other instructions. The translation of a few other constructs is worth discussing.

- *Assertions.* Java's *assert exp* raises an exception if *exp* evaluates to **false**, whereas a failed **check** *exp* **end** in Eiffel sends a signal to the runtime which terminates execution or invokes the debugger. Java's assertions

are therefore translated to:
if not *exp* **then** ∇_M(**throw** (**new** *AssertionError* ())) **end**

- *Block locals* are moved to the beginning of the current method; the naming scheme (Section 5.3.4) prevents name clashes.

- *Calls to parent's methods.* Eiffel's **Precursor** can only invoke the parent's version of the overridden routine currently executed, not any feature of the parent. The translation $\mathcal{T}$ augments every method with an extra boolean argument *predecessor* and calls **Precursor** when invoked with *predecessor* set to **True**; this accommodates any usage of **super**:

```
∇B(type method(args) {body})   =   method (args ; predecessor: BOOLEAN): ∇T(type)
                                     do
                                       if predecessor then
                                         Precursor (args, False)
                                       else
                                         ∇M(body)
                                       end
                                     end
∇E(method(exp))                =   method (∇E(exp), False)
∇E(super.method(exp))          =   method (∇E(exp), True)
```

- *Casting and type conversions* are adapted to Eiffel with the services provided by the helper class *JAVA_TYPE_HELPER*.

- *Expressions used as instructions* are wrapped into the helper routine *dev_null* (*a*: *ANY*): ∇_M(exp) = *dev_null* (∇_E (exp)).

- *Switch statements* become **if**..**elseif**..**else** blocks in Eiffel, nested within a loop to support fall-through.

How J2Eif implements $\mathcal{T}$.

As a single example of how the implementation of $\mathcal{T}$ deviates from the formal presentation, consider J2Eif's translation of exception-handling blocks **try**{doTry} **catch**(*t e*){doCatch} **finally**{doFinally}:

```
skipFlag := False
rethrowFlag := False
(agent (args)
   do
      if not skipFlag then
         ∇M(doTry)
      else
         if e.conforms_to (∇T(t)) then
            ∇M(doCatch)
         else
            rethrowFlag := True
         end
      end
```

```
    skipFlag := True
    ∇M(doFinally)
    if rethrowFlag and not breakFlag then
      (create {EXCEPTION}).raise
    end
  rescue
    if not skipFlag then
      skipFlag := True
      Retry := True
    end
  end).call
```

This translation applies uniformly to all exception-handling code and avoids duplication of the **finally** block, hence the *agent*'s body structure is more similar to the Java source. The formalization ∇_M above, however, allows for a more focused presentation and lends itself to easier formal reasoning (see Section 5.4.1). A correctness proof of the implementation could then establish that ∇_M and the implementation J2Eif describe translations with the same semantics.

5.3.2 Built-in Types

The naming scheme (Section 5.3.4) handles references to classes and interfaces as types; primitive types and some other type constructors are discussed here.

- *Primitive types* with the same machine size are available in both Java and Eiffel: Java's **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**, and **double** exactly correspond to Eiffel's *BOOLEAN*, *CHARACTER_32*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *REAL_32*, and *REAL_64*.
- *Arrays* in Java become instances of Eiffel's helper class *JAVA_ARRAY*, which inherits from the standard EiffelBase *ARRAY* class and completes it with all missing Java operations.
- *Annotations and enumerations* are syntactic sugar for interfaces and classes respectively extending *java.lang.annotation.Annotation* and *java.lang.Enum*.

5.3.3 Runtime and Native Interface

This section describes how J2Eif replicates, in Eiffel, JRE's functionalities.

Reflection.

Compared to Java, Eiffel has only limited support for reflection and dynamic loading. The translation $\mathcal{T}$ ignores dynamic loading and includes all classes

required by the system for compilation. The translation itself also generates reflection data about every class translated and adds it to the produced Eiffel classes; the data includes information about the parent class, fields, and methods, and is stored as objects of the helper class *JAVA_CLASS*. For example, $\mathcal{T}$ generates the routine *get_class* for *JAVA_LANG_STRING_STATIC*, the Eiffel counterpart to the static component of *java.lang.String*, as follows:

```
get_class: JAVA_CLASS
    once ("PROCESS")
        create Result.make ("java.lang.String")
        Result.set_superclass (create {JAVA_LANG_OBJECT_STATIC})
        Result.fields.extend (["count" field data])
        Result.fields.extend (["value" field data])
        ...
        Result.methods.extend (["equals" method data]))
        ...
    end
```

Concurrency.

J2Eif includes a modified translation of *java.lang.Thread* which inherits from the Eiffel *THREAD* class and maps Java threads' functionalities to Eiffel threads; for example, the method *start*() becomes a call to the routine *launch* of class *THREAD*. *java.lang.Thread* is the only JRE library class which required a slightly *ad hoc* translation; all other classes follow the general scheme presented in the present chapter.

Java's **synchronized** methods work on the implicit *monitor* associated with the current object. The translation to Eiffel adds a *mutex* attribute to every class which requires synchronization, and explicit locks and unlocks at the entrance and exit of every translated **synchronized** method:

∇_B(**synchronized** type *method*(args){body}) = *method* (args): ∇_T(type)
do
mutex.lock
∇_M(*body*)
mutex.unlock
end

Native interface.

Java Native Interface (JNI) supports calls between pre-compiled libraries and Java applications. JNI is completely independent of the rest of the Java runtime: a C **struct** includes, as function pointers, all references to native methods available through the JNI. Since Eiffel includes an extensive support to call external C code through the CECIL library, replicating JNI's

functionalities in J2Eif is straightforward. The helper class *JAVA_PARENT*—accessible in every translated class—offers access to a **struct** *JNIEnv*, which contains function pointers to suitable functions wrapping the native code with CECIL constructs. This way, the Eiffel compiler is able to link the native implementations to the rest of the generated binary.

This mechanism works for all native JRE libraries except for the Java Virtual Machine (*jvm.dll* or *jvm.so*), which is specific to the implementation (OpenJDK in our case) and had to be partially re-implemented for usage within J2Eif. The current version includes new implementations of most JVM-specific services, such as *JVM_FindPrimitiveClass* to support reflection or *JVM_ArrayCopy* to duplicate array data structures, and verbatim replicates the original implementation of all native methods which are not JVM-specific (such as *JVM_CurrentTimeMillis* which reads the system clock). The experiments in Section 5.4 demonstrate that the current JVM support in J2Eif is extensive and sufficient to translate correctly many Java applications.

Garbage collector.

The Eiffel garbage collector is used without modifications; the marshalling mechanism can also collect JNI-maintained instances.

5.3.4 Naming

The goal of the renaming scheme introduced in the translation $\mathcal{T}$ is threefold: to conform to Eiffel's naming rules, to make the translation as readable as possible (i.e., to avoid cumbersome names), and to ensure that there are no name clashes due to different conventions in the two languages (for example, Eiffel is completely case-insensitive and does not allow in-class method overload).

To formalize the naming scheme, consider the functions η, ϕ, and λ:

- η normalizes a name by successively (1) replacing all "_" with "_1", (2) replacing all "." with "_", and (3) changing all characters to uppercase—for example, η(*java.lang.String*) is *JAVA_LANG_STRING*;

- $\phi(n)$ denotes the fully-qualified name of the item n—for example, ϕ(*String*) is, in most contexts, *java.lang.String*;

- $\lambda(v)$ is an integer denoting the nesting depth of the block where v is declared—in the method **void** *foo*(**int** *a*){**int** *b*; **for**(**int** *c*=0;...)...} for example, it is $\lambda(a) = 0$, $\lambda(b) = 1$, $\lambda(c) = 2$.

Then, the functions $\Delta_C, \Delta_F, \Delta_M, \Delta_L$ respectively define the renaming scheme for class/interface, field, method, and local name; they are defined as follows, where $\oplus$ denotes string concatenation, "className" refers to the name of the class of the current entity, and ϵ is the empty string.

$$
\begin{array}{ll}
\Delta_C(\text{className}) & = \eta(\phi(\text{className})) \\
\Delta_F(\text{fieldName}) & = \text{“}\textit{field\_}\text{”} \oplus \lambda(\text{fieldName}) \oplus \text{“\_”} \oplus \text{fieldName} \oplus \text{“\_”} \oplus \Delta_C(\text{className}) \\
\Delta_L(\text{localName}) & = \text{“}\textit{local\_}\text{”} \oplus \lambda(\text{localName}) \oplus \text{“\_”} \oplus \text{localName} \\
\Delta_M(\text{className(args)}) & = \text{“}\textit{make\_}\text{”} \oplus \Delta_A(\text{args}) \oplus \Delta_C(\text{className}) \\
\Delta_M(\text{methodName(args)}) & = \text{“}\textit{method\_}\text{”} \oplus \text{methodName} \oplus \Delta_A(\text{args}) \\
\Delta_A(\text{t}_1\ \text{n}_1, \ldots, \text{t}_m\ \text{n}_m) & = \begin{cases} \epsilon & \text{if } m = 0 \\ \text{“}\textit{from\_}\text{”} \oplus \delta(\text{t}_1) \oplus \ldots \oplus \delta(\text{t}_m) & \text{if } m > 0 \end{cases} \\
\delta(\text{t}) & = \begin{cases} \text{“}p\text{”} \oplus \text{t} & \text{if t is a primitive type} \\ \text{t} & \text{otherwise} \end{cases}
\end{array}
$$

The naming scheme renames classes to include their fully qualified name. It labels fields and appends to their name their nesting depth (higher than one for nested classes) and the class they belong to; similarly, it labels locals and includes their nesting depth in the name. It pre-pends "*make*" to constructors—whose name in Java coincides with the class name—and "*method*" to other methods. To translate overloaded methods, it includes a textual description of the method's argument types to the renamed name, according to function Δ_A; an extra *p* distinguishes primitive types from their boxed counterparts (e.g., **int** and *java.lang.Integer*). Such naming scheme for methods does not use the fully qualified name of argument types. This favors the readability of the names translated over certainty of avoiding name clashes: a class may still overload a method with arguments of different type but sharing the same unqualified name (e.g., *java.util.List* and *org.eclipse.Swt.Widgets.List*). This, however, is extremely unlikely to occur in practice, hence the chosen trade-off is reasonable.

5.4 Evaluation

This section briefly discusses the correctness of the translation $\mathcal{T}$ (Section 5.4.1); evaluates the usability of its implementation J2Eif with four case studies (Section 5.4.2); and concludes with a discussion of open issues (Section 5.4.3).

5.4.1 Correctness of the Translation

While the formalization of $\mathcal{T}$ in the previous sections is not complete and overlooks some details, it is useful to present the translation clearly, and it even helped the authors find a few errors in the implementation when its results did not match the formal model. Assuming an operational semantics

for Java and Eiffel (see [36]), one can also reason about the components of $\mathcal{T}$ formalized in Section 5.3 and increase the confidence in the correctness of the translation. This section gives an idea of how to do it; a more accurate analysis would leverage a proof assistant to ensure that all details are taken care of appropriately.

The operational semantics defines the effect of every instruction I on the program state: $\sigma \xrightarrow{I} \sigma'$ denotes that executing I on a state σ transforms the state to σ'. The states σ, σ' may also include information about exceptions and non-terminating computations. While a Java and an Eiffel state are in general different, because they refer to distinct execution models, it is possible to define an equivalence relation $\simeq$ that holds for states sharing the same "abstract" values [36], which can be directly compared. With these conventions, it is possible to prove correctness of the formalized translation: the effect of executing a translated Eiffel instruction on the Eiffel state replicates the effect of executing the original Java instruction on the corresponding Java state. Formally, the correctness of the translation of a Java instruction I is stated as: "For every Java state σ_J and Eiffel state σ_E such that $\sigma_J \simeq \sigma_E$, if $\sigma_J \xrightarrow{I} \sigma'_J$ and $\sigma_E \xrightarrow{\nabla_M(I)} \sigma'_E$ then $\sigma'_J \simeq \sigma'_E$."

The proof for the the Java block B: **try** {doTry} **catch** (t e) {doCatch}, translated to $\nabla_M(B)$ as shown on page 75, is now sketched. A state σ is split into two components $\sigma = \langle v, e \rangle$, where e is ! when an exception is pending and $\star$ otherwise. The proof works by structural induction on B; all numeric references are to Nordio's operational semantics [36, Chap. 3]; for brevity, consider only one inductive case.

- *doTry raises an exception handled by doCatch*: $\langle v_J, \star \rangle \xrightarrow{\text{doTry}} \langle v'_J, ! \rangle$, the type τ of the exception raised conforms to t, and $\langle v'_J, ! \rangle \xrightarrow{\text{doCatch}} \langle v''_J, e \rangle$, hence $\langle v_J, \star \rangle \xrightarrow{B} \langle v''_J, e \rangle$ by (3.12.4). Then, both $\langle v_E, \star \rangle \xrightarrow{\nabla_M(\text{doTry})} \langle v'_E, ! \rangle$ and $\langle v'_E, ! \rangle \xrightarrow{\nabla_M(\text{doCatch})} \langle v''_E, e' \rangle$ hold by induction hypothesis, for some $v'_E \simeq v'_J$, $v''_E \simeq v''_J$, and $e' \simeq e$. Also, *e.conforms_to* ($\nabla_T(t)$) evaluates to false on the state v'_E. In all, $\langle v_E, \star \rangle \xrightarrow{\nabla_M(B)} \langle v''_E, e' \rangle$ by (3.10) and the rule for **if**..**then**.

5.4.2 Experiments

Table 5.2 shows the results of four experiments run with J2Eif on a Windows Vista machine with a 2.66 GHz Intel dual-core CPU and 4 GB of memory. Each experiment consists in the translation of a program (stand-alone application or library). Table 5.2 reports: (1) the size in lines of code of the source (J for Java) and transformed program (E for Eiffel); (2) the size

	Size (locs)		#Classes		Time	Binary Size	#Required
	J	E	J	E	(sec.)	(MB)	Classes
HelloWorld	5	92	1	2	1	65	1,208
SWT snippet	34	313	1	6	47	88	1,208 (317)
java.util.∗	51,745	91,162	49	426	7	65	1,175
java.io tests	11,509	28,052	123	302	6	65	1,225

Table 5.2: Experimental results.

in number of classes; (3) the source-to-source translation time (in seconds), which does not include the compilation from Eiffel source to binary; (4) the size (in MBytes) of the binaries generated by EiffelStudio; (5) the number of dependent classes needed for the compilation (the SWT snippet entry also reports the number of SWT classes in parentheses). The rest of the section discusses the experiments in more detail.

HelloWorld. The *HelloWorld* example is useful to estimate the minimal number of dependencies included in a stand-alone application in order to emulate the Java runtime. The size of 65 MB is the smallest footprint of any application generated with J2Eif.

SWT snippet. The SWT snippet generates a window with a browsable calendar and a clock. While simple, the example demonstrates that J2Eif correctly translates GUI applications and replicates their behavior: this enables Eiffel programmers to include in their programs services from libraries such as SWT.

***java.util.∗* classes.** Table 5.3 reports the results of performance experiments on some of the translated versions of the 49 data structure classes in *java.util*. For each Java class with an equivalent data structure in EiffelBase, we performed tests which add 100 elements to the data structure and then perform 10000 removals of an element which is immediately re-inserted. Table 5.3 compares the time (in ms) to run the test using the translated Java classes (column 2) to the performance with the native EiffelBase classes (column 4).

The overhead introduced by some features of the translation adds up in the tests and generates the significant overall slow-down shown in Table 5.3. The features that most slowed down the translated code are: (1) the indirect access to fields via the *JAVA_VARIABLE* class; (2) the more structured (and slower) translation of control-flow breaking instructions; (3) the handling of exceptions with agents (whose usage is as expensive as a method

Java class	Translated Java time	Eiffel class	Eiffel time	Slowdown
ArrayList	582	*ARRAYED_LIST*	139	4.2
Vector	620	*ARRAYED_LIST*	139	4.5
HashMap	1,740	*HASH_TABLE*	58	30
Hashtable	1,402	*HASH_TABLE*	58	24.2
LinkedList	560	*LINKED_LIST*	94	6
Stack	543	*ARRAYED_STACK*	26	20.9

Table 5.3: Performance of translated *java.util* classes.

	Overall time (s)	Average time per test (ms)	Slowdown
Java	4	5	1
Eiffel	9	11	2.2

Table 5.4: Performance in the *java.io* test suite.

call). Applications that do not heavily exercise data structures (such as GUI applications) are not significantly affected and do not incur a nearly as high overhead.

***java.io* test suite.** The part of the Mauve test suite [27] focusing on testing input/output services consists of 102 classes defining 812 tests. The tests with J2Eif excluded 10 of these classes (and the corresponding 33 tests) because they relied on unsupported features (see Section 5.4.3). The functional behavior of the tests is identical in Java and in the Eiffel translation: both runs fail 25 tests and pass 754. Table 5.4 compares the performance of the test suite with Java against its Eiffel translation; we consider the two-fold slowdown usable and reasonable—at least in a first implementation of J2Eif.

5.4.3 Limitations

There is a limited number of features which J2Eif does not handle adequately; ameliorating them belongs to future work.

- *Unicode strings.* J2Eif only supports the ASCII character set; Unicode support in Eiffel is quite recent.
- *Serialization* mechanisms are not mapped adequately to Eiffel's.
- *Dynamic loading* mechanisms are not rendered in Eiffel; this restricts the applicability of J2Eif for applications heavily depending on this mechanism, such as J2Eif itself which builds on the Eclipse framework.

- *Soft, weak, and phantom references* are not supported, because similar notions are currently not available in the Eiffel language.
- *Readability.* While the naming scheme tries to strike a good balance between readability and correctness, the generated code may still be less pleasant to read than in a standard Eiffel implementation.
- *Size of compiled code.* The generated binaries are generally large. A finer-grained analysis of the dependencies may reduce the JRE components that need to be included in the compilation.

5.5 Related Work

As far as fully automatic translations are concerned, compilation from a high-level language to a low-level language (such as assembly or byte-code) is of course a widespread technology. The translation of a high-level language into another high-level language with different features—such as the one performed by J2Eif—is much less common; the closest results have been in the rewriting of domain-specific languages, such as TXL [4], into general-purpose languages.

Google web toolkit [15] (GWT) includes a project involving translation of Java into JavaScript code. The translation supports running Java on top of JavaScript, but its primary aims do not include readability and modifiability of the code generated, unlike the present chapter's translation. Another relevant difference is that GWT's translation lacks any formalization and even the informal documentation does not detail which features are not perfectly replicated by the translation. The documentation warns the users that "subtle differences" may exist,[1] but only recommends testing as a way to discover them.

5.6 Summary

This chapter presented a translation of Java programs into Eiffel, and its implementation in the freely available tool J2Eif [17]. The translation has been formalized in order to increase confidence in its correctness, and the usability of J2Eif has been evaluated on a set of four programs of varying complexity.

[1] `http://code.google.com/webtoolkit/doc/latest/tutorial/JUnit.html`

Chapter 6

Conclusions

This dissertation includes multiple contributions to the state of the art in programming language translation and object-oriented reengineering. We have developed a technique for automatic migration of program source code from C, a legacy programming language, into Eiffel, a modern object-oriented environment. The technique has been implemented in a freely available tool C2Eif.

We have also proposed an approach to object-oriented reengineering of C programs, which we implemented as a C2Eif extension, called AutoOO. We confirmed the soundness and practicality of both techniques with an extensive evaluation on a number of real-world applications and libraries, including the editor `vim` and the math library `libgsl`. Our translation produces object-oriented code with high level of encapsulation and introduces inheritance, contracts, and exceptions where appropriate.

Lastly, we have investigated the feasibility of translating code between modern object-oriented environments and developed J2Eif: a fully automatic translator from Java into Eiffel.

There are two major differences between the present work and existing approaches to code migration and object-oriented reengineering. First, our translation process is *completely automatic*: it produces ready-to-compile, functionally equivalent target code, without any human intervention. Second, our tools and techniques support the *full source language*: they do not shy away from the trickier aspects of the translation and reengineering – be it pointer arithmetic, calls to external libraries or creation of instance routines – which makes them applicable to real-world code.

6.1 Future Work

The main focus of the dissertation has been to investigate to what extent a fully automatic translation can be achieved. While we paid considerable attention to generating readable and understandable code, we believe there is still potential for improvement in this area.

Our technique for object-oriented reengineering might be overly conservative in some cases. We do not perform a refactoring unless it is guaranteed to preserve the program behavior and is expected, with high confidence, to be sound. Future work should investigate the cases where this approach is too restrictive and extend the reengineering to code elements (such as bundle classes) that are currently kept unmodified.

Bibliography

[1] B. L. Achee and Doris L Carver. Creating object-oriented designs from legacy FORTRAN code. *Journal of Systems and Software*, 39(2):179–194, 1997.

[2] V.H. Allan and X. Chen. Convert2java: semi-automatic conversion of c to java. *Future Generation Computer Systems*, 18(2):201–211, 2001. Java in HPC.

[3] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[4] James R. Cordy. Source transformation, analysis and generation in TXL. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 1–11, 2006.

[5] Crispin Cowan. FormatGuard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[6] CWE-134. Uncontrolled format string. `http://cwe.mitre.org/data/definitions/134.html`.

[7] Andrea de Lucia, Giuseppe A. Di Lucca, Anna Rita Fasolino, Patrizia Guerra, and Silvia Petruzzelli. Migrating legacy systems towards object-oriented platforms. pages 122–129, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[8] W. C. Dietrich, Jr., L. R. Nackman, and F. Gracer. Saving legacy with objects. *SIGPLAN Not.*, 24(10):77–83, 1989.

[9] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 533–544, 2012.

[10] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[11] R. Fanta and V. Rajlich. Reengineering object-oriented code. In *Proceedings of the International Conference on Software Maintenance*, pages 238–246, 1998.

[12] FFI. A portable foreign function interface library. `http://sources.redhat.com/libffi/`, 2011.

[13] William Frakes, Gregory Kulczycki, and Natasha Moodliar. An empirical comparison of methods for reengineering procedural software systems to object-oriented systems. In *Proceedings of the 10th international conference on Software Reuse: High Confidence Software Reuse in Large Systems*, volume 5030 of *LNCS*, pages 376–389, 2008.

[14] H. Gall and R. Klosch. Finding objects in procedural programs: an alternative approach. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 208–216. IEEE, 1995.

[15] Google Web toolkit. `http://code.google.com/webtoolkit/`, 2010.

[16] David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, 1980.

[17] J2Eif. The Java to Eiffel translator. `http://se.inf.ethz.ch/research/j2eif/`, 2010.

[18] Ivar Jacobson and Fredrik Lindström. Reengineering of old systems to an object-oriented architecture. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 340–350. ACM, 1991.

[19] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code migration through transformations: an experience report. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 201–213, Riverton, NJ, USA, 2010. IBM Corp.

[20] Kostas Kontogiannis and Prashant Patil. Evidence driven object identification in procedural code. In *Software Technology and Engineering Practice, 1999*, pages 12–21, 1999.

[21] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*, pages 470–485. Springer, 2009.

[22] Panos E. Livadas and Theodore Johnson. A new approach to finding objects in programs. *Journal of Software Maintenance*, 6(5):249–260, 1994.

[23] Johannes Martin. *Ephedra - A C to Java Migration Environment*. PhD thesis, Department of Computer Science, University of Victoria, 2002.

[24] Johannes Martin and Hausi A. Müller. *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*, chapter Discovering implicit inheritance relations in non object-oriented code, pages 177–193. Springer Verlag, New York, May 2001.

[25] Johannes Martin and Hausi A. Müller. Strategies for migration from C to Java. In *Fifth European Conference on Software Maintenance and Reengineering*, pages 200–210. IEEE Computer Society, 2001.

[26] Johannes Martin and Hausi A. Müller. C to Java migration experiences. In *Sixth European Conference on Software Maintenance and Reengineering*, pages 143–153. IEEE Computer Society, 2002.

[27] Mauve project. `http://sources.redhat.com/mauve/`, 2010.

[28] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[29] M. Mossienko. Automated Cobol to Java recycling. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 40–50. IEEE, 2003.

[30] P. Müller and M. Nordio. Proof-Transforming Compilation of Programs with Abrupt Termination. In *Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, 2007.

[31] Beevi S. Nadera, D. Chitraprasad, and Vinod S. S. Chandra. The varying faces of a program transformation systems. *ACM Inroads*, 3(1):49–55, mar 2012.

[32] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Conference on Compiler Construction*, pages 213–228, 2002.

[33] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[34] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, 2007.

[35] P. Newcomb and G. Kotik. Reengineering procedural into object-oriented systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 237–249, 1995.

[36] Martin Nordio. *Proofs and Proof Transformations for Object-Oriented Programs*. PhD thesis, ETH Zurich, 2009.

[37] Novosoft. C2J: a C to Java translator. `http://www.novosoft-us.com/solutions/product_c2j.shtml`, 2001.

[38] Yutaka Oiwa. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 259–269, New York, NY, USA, 2009. ACM.

[39] Margot Postema and Heinz W. Schmidt. Reverse engineering and abstraction of legacy systems. *Informatica*, pages 37–55, 1998.

[40] Dennis Ritchie. The development of the C language. In *History of Programming Languages Conference (HOPL-II) Preprints*, pages 201–208, 1993.

[41] Miguel A. Serrano, Doris L. Carver, and Carlos Montes de Oca. Reengineering legacy systems for distributed environments. *Journal of Systems and Software*, 64(1):37–55, 2002.

[42] H. Sneed. Migrating PL/I code to Java. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, pages 287–296. IEEE, 2011.

[43] H.M. Sneed. Migration of procedurally oriented Cobol programs in an object-oriented architecture. In *Proceerdings of the 1992 Conference on Software Maintenance*, pages 105–116, 1992.

[44] H.M. Sneed. Planning the reengineering of legacy systems. *IEEE Software*, 12(1):24–34, jan 1995.

[45] H.M. Sneed. Migrating from COBOL to Java. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, pages 1–7. IEEE, 2010.

[46] Gokul V. Subramaniam and Eric J. Byrne. Deriving an object model from legacy Fortran code. *Proceedings of the 1996 International Conference on Software Maintenance*, pages 3–12, 1996.

[47] Ricky Sward. Extracting ada 95 objects from legacy ada programs. In *Reliable Software Technologies - Ada-Europe 2004*, volume 3063 of *Lecture Notes in Computer Science*, pages 65–77. Springer, 2004.

[48] Tangible Software Solutions. C++ to C# and C++ to Java. `http://www.tangiblesoftwaresolutions.com/`.

[49] Andrey A. Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.

[50] Eli Tilevich. Translating C++ to Java. In *German Java Developers' Conference Journal*. Sun Microsystems Press, 1997.

[51] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 191–200. ACM, 2011.

[52] Theo Wiggerts, Hans Bosma, and Erwin Fielt. Scenarios for the identification of objects in legacy systems. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 24–32, 1997.

[53] A. Yeh, D. Harris, and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 227–236, 1995.

[54] Ying Zou and Kostas Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *Proceedings of the Eighth Asia-Pacific Software Engineering Conference*, pages 390–399, 2001.

CURRICULUM VITAE

General Information

Name: Marco Trudel
Date of birth: July 18, 1983
Nationality: Swiss
Web page: `http://se.inf.ethz.ch/people/trudel`
E-mail address: marco.trudel@inf.ethz.ch

Education

- January 2009 – present, ETH Zürich
  PhD student at the Chair of Software Engineering, Department of Computer Science

- October 2006 – December 2008, ETH Zürich
  Master of Science in Computer Science, major in Software Engineering

- October 2003 – November 2006, Zürich University of Applied Sciences (ZHAW)
  Bachelor of Engineering, major in Software Engineering

- August 1999 – August 2003, Viviance AG and Avantix AG
  Computer Science Apprenticeship

Positions

- January 2009 – present, ETH Zürich
  Research assistant at the Chair of Software Engineering, Department of Computer Science

- February 2006 – April 2006, Avallain AG
  Software Engineer (Freelancer)
- August 2003 – November 2004, Avantix AG
  Software Engineer and System Administrator

Publications

- M. Trudel, C. A. Furia, M. Nordio, B. Meyer, *Really Automatic Scalable Object-Oriented Reengineering*, ECOOP 2013
- M. Trudel, C. A. Furia, M. Nordio, B. Meyer, M. Oriol, *C to O-O Translation: Beyond the Easy Stuff*, WCRE 2012
- M. Trudel, C. A. Furia, M. Nordio, *Automatic C to O-O Translation with C2Eiffel*, WCRE 2012, tool demonstration paper
- M. Trudel, M. Oriol, C. A. Furia, M. Nordio, *Automated Translation of Java Source Code to Eiffel*, TOOLS Europe 2011