C to O-O Translation: Beyond the Easy Stuff[*]

Marco Trudel · Carlo A. Furia · Martin Nordio
Bertrand Meyer · Manuel Oriol

19 April 2013

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 paper presents a source-to-source translation of C code into Eiffel, a modern object-oriented programming language, and the supporting tool C2Eif. The translation is completely automatic and handles the entire C language (ANSI, as well as many GNU C Compiler extensions, through CIL) as used in practice, including its usage of native system libraries and inlined assembly code. 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 translations.

1   Introduction

Programming languages have significantly evolved since the original design of C in the 1970's as a "system implementation language" [29] for the Unix operating system. C was a high-level language by the standards of the time, but it is pronouncedly low-level compared with modern programming paradigms, as it lacks advanced features—static type safety, encapsulation, inheritance, and contracts [20], to mention just a few—that can have a major impact on programmer's productivity and on software quality and maintainability.

C still fares as the most popular general-purpose programming language [38], and countless C applications are being actively written and maintained, that take advantage of the language's conciseness, speed, ubiquitous support, and huge code-base. An automated solution to translate and integrate C code into a modern language would combine the large availability of C programs in disparate domains with the integration in a modern language that facilitates writing safe, robust, and easy-to-maintain applications.

The present paper describes the fully automatic translation of C programs into Eiffel, an object-oriented programming language, and the 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" (see Section 5 for examples), source-to-source translation solves a different problem, and has some distinctive benefits: the translated code can take 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 review in Section 5). A limitation of the resulting tools is that they hardly handle the trickier or specialized parts of the C language [7], which it is 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 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 1, and a user guide.

Sections 2–3 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 [22]; 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 strong typing and contracts, to facilitate debugging some 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, and is the main motivation behind this work. However, it also implies other valuable side-benefits, which we demonstrate in Section 4. First, the translated code blends reasonably 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

---

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

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 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 into 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.

Why Eiffel? We chose Eiffel as the target language not only out of our familiarity with it, but also because it offers features that complement C's, such as an emphasis on correctness [44] through the native support of contracts.

Another reason for the choice has to do with the size of Eiffel's community of developers, which is quite small compared to those of other object-oriented languages such as Java or C#. While Java users can avail of countless libraries and frameworks offering high-quality implementations of a wide array of functionalities, Eiffel users often have to implement such functionalities themselves from scratch. An automatic translation tool could therefore bring immediate benefits to reuse the rich C code-base in Eiffel; in fact, translations produced by C2Eif are already being used by the Eiffel community.

Finally, Eiffel uncompromisingly epitomizes the object-oriented paradigm; hence translating C into it cannot take the shortcut of merely transliterating similar constructs (as it would have been possible, for example, with C++). The results of the paper are thus applicable to other object-oriented languages.[2]

An abridged version of this paper was presented at WCRE 2012 [42]; a companion paper presents the tool C2Eif from the user's point of view [40].

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

High-level view. Figure 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 [22] 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 (although it also introduces some limitations, discussed in Section 4.5). C2Eif then translates CIL programs to Eiffel projects

---

[2]As proof of this, the first author has made some progress towards applying the technique described in this paper to automatic translations from C to Java; and has recently been granted startup funding to support this development and turn it into a commercial product.
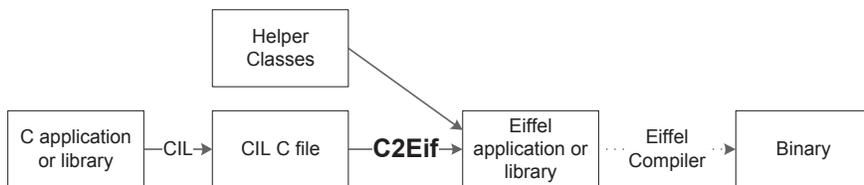
Figure 1: Usage workflow of C2Eif.

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 45 such transformations (i.e., $n = 45$). 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 involves only a limited number of classes. We now give a concise description of the most important helper classes; Section 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 which 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).

4

- *CE_CLASS* defines the basic interface of Eiffel classes that correspond to **union**s and **struct**s. It includes features (members) 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 the 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 Translating C to Eiffel

This section presents the major details 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 down $\mathcal{T}$ into several components that target different language aspects (for example, $\mathcal{T}_{\mathrm{TD}}$ maps C type declarations to Eiffel classes); these components mirror the incremental transformations T_i of C2Eif (mentioned in Section 2) but occasionally overlook inessential details for presentation clarity.

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[3] or call functions declared in header files. For example, the following Eiffel routine (method) *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 2) includes an attribute *c_pointer* of type *POINTER* that offers access to a C-conforming representation usable in **external** calls.

---

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

The mechanism of external calls is used not only in the translations produced by C2Eif but also in the Eiffel standard libraries, wherever interaction with the operating system is required, such as for input/output, file operations, and memory management. Supporting calls to native code is also the only strict requirement to be able to implement, in languages other than Eiffel, this paper's approach to automatic translation from C. Java, for example, offers similar functionalities through the Java Native Interface (JNI).

3.1 Types and Type Constructors

C declarations $T\ v$ of a variable v of type T become Eiffel declarations $v : \mathcal{T}_{\mathrm{TY}}(T)$, where $\mathcal{T}_{\mathrm{TY}}$ 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 numbers) with the appropriate bit-size as follows[4]:

C TYPE T	EIFFEL CLASS $\mathcal{T}_{\mathrm{TY}}(T)$
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 $\mathcal{T}_{\mathrm{TY}}($**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:

$$\mathcal{T}_{\mathrm{TY}}(T *) \quad = CE\_POINTER\ [\mathcal{T}_{\mathrm{TY}}(T)]$$

with the convention that $\mathcal{T}_{\mathrm{TY}}($**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 $\mathcal{T}_{\mathrm{TY}}($**float** $**)$.

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

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

CE_ROUTINE inherits from *CE_POINTER* [*ANY*], and hence it behaves as a generic pointer, but customizes its behavior using references to **agent**s that wrap the functions pointed to; Section 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: $\mathcal{T}_{\mathrm{TY}}(T\ [n]) = CE\_ARRAY[\mathcal{T}_{\mathrm{TY}}(T)]$.

---

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

The size parameter n, if present, does not affect the declaration, but initializations of array variables use it (see Section 3.2). Multi-dimensional arrays are defined recursively as arrays of arrays: $\mathcal{T}_{\mathrm{TY}}(T\,[n_1][n_2]...[n_m])$ is then $CE\_ARRAY[\mathcal{T}_{\mathrm{TY}}(T[n_2]...[n_m])]$.

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

$\mathcal{T}_{\mathrm{TD}}(\textbf{enum } E \,\{v_1 = k_1, \ldots, v_m = k_m\}) =$

class *E* **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, $\mathcal{T}_{\mathrm{TY}}(\textbf{enum } E) = INTEGER\_32$), and the class *E* is only used to assign constant values according to the **enum** naming scheme.

Structs and unions. For every compound type **struct** *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* which also updates the underlying C representation of *v*:

v: $\mathcal{T}_{\mathrm{TY}}(T)$ **assign** *set_v* −− declares 'set_v' as the setter of v
set_v (*a_v*: $\mathcal{T}_{\mathrm{TY}}(T)$) **do** *v* := *a_v* ; *update_memory_field* ("*v*") **end**

Class *CE_CLASS*, from which *S* inherits, 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.

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 $\mathcal{T}_{\mathrm{TD}}$ 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

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.

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 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** $a.make(\ll 2, 3 \gg)$.

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 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** $p.make\_cast\,(malloc\,(\sigma(T)))$
$free(p)$;	$free(p)$

8

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 *CE_ARRAY*, *CE_POINTER*, and classes translating C **struct**s and **union**s, 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'** using 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 and pointer arithmetic. The helper class *CE_POINTER* features a query *item* which 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 an integer variable *num*, a pointer variable *p*, and a double pointer variable *carsp* with target type **struct** *car* (defined in Example 1). The following C code fragment declares *num*, *p*, and *carsp*, allocates space for an array of *num* consecutive cars pointed to by *carsp*, and makes *p* point to the array's third element.

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

C2Eif translates the C fragment as follows, where the implicit C conversion from signed to unsigned **int** become explicit in Eiffel (calls to routine *to_natural_32*).

```
num: INTEGER_32
p: CE_POINTER [CAR]
carsp: CE_POINTER [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
```
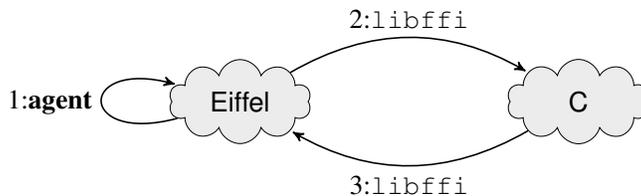
Figure 2: Calls through function pointers.

Using function pointers. Class *CE_ROUTINE*, which translates C function pointers, is usable both in the Eiffel and in the C environment (see Figure 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` [9].

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 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 2).

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 2) and dynamically converted to suitable **agent** invocations.

3.3 Conditionals, Loops, and Functions

The translation $\mathcal{T}_{\mathrm{CF}}$ takes care of constructs to structure the control flow of computations: conditionals, loops, and function definitions and calls.

Conditionals and loops. Sequential composition, conditional instructions, and loop instructions are similar in C and Eiffel; therefore, their translation is straightforward.[5]

---

[5]Eiffel loops have the general form: **from** *init* **until** *exit* **loop** *B* **end**. *init* is executed once before the ac-

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

Note that Eiffel has only one type of loop, which translates the unique type of C loops produced by CIL.

Functions. Function declarations and function calls in C translate to routine declarations and routine calls in Eiffel. Given a function *foo* with arguments $a_1, \ldots, a_n$ of type $T_1, \ldots, T_n$ and return type T_0, the translation:

$$
\mathcal{T}_{\mathrm{CF}}\big(\ T_0\,foo\ (T_1\ a_1, \ldots, T_n\ a_n)\{\ B\ \}\ \big)
$$

is defined as follows:

$$
\begin{cases}
foo(a_1\!:\!\mathcal{T}_{\mathrm{TY}}(T_1); \ldots; a_n\!:\!\mathcal{T}_{\mathrm{TY}}(T_n))\ \textbf{do } \mathcal{T}(B)\ \textbf{end} & \text{if } T_0 \text{ is } \textbf{void} \\
foo(a_1\!:\!\mathcal{T}_{\mathrm{TY}}(T_1); \ldots; a_n\!:\!\mathcal{T}_{\mathrm{TY}}(T_n)) : \mathcal{T}_{\mathrm{TY}}(T_0)\ \textbf{do } \mathcal{T}(B)\ \textbf{end} & \text{otherwise}
\end{cases}
$$

Correspondingly, function calls in C become routine calls in Eiffel with the actual arguments also recursively translated:

$$
\mathcal{T}_{\mathrm{CF}}(foo\ (e_1, \ldots, e_n)) \ = \ foo\ (\mathcal{T}(e_1), \ldots, \mathcal{T}(e_n))
$$

Variadic functions. The C language supports *variadic functions*, also known as *varargs functions*, which work on a variable number of arguments. The Eiffel translation of variadic function declarations uses the class *TUPLE* to wrap lists of arguments: the translation of a variadic function *var_foo* with $n \geq 0$ required arguments $a_1, \ldots, a_n$ of type $T_1, \ldots, T_n$, optional additional arguments, and return type T_0:

$$
\mathcal{T}_{\mathrm{CF}}\big(\ T_0\ var\_foo\ (T_1\ a_1, \ldots, T_n\ a_n, \ldots)\{\ B\ \}\ \big)
$$

is defined as:

$$
var\_foo\ (args\colon TUPLE[a_1\colon \mathcal{T}_{\mathrm{TY}}(T_1); \ \ldots; \ a_n\colon \mathcal{T}_{\mathrm{TY}}(T_n)])\colon \mathcal{T}_{\mathrm{TY}}(T_0)
$$

assuming T_0 is not **void**; otherwise, the return type is omitted, as in the translation of standard functions.

Calls to variadic functions with n required arguments may provide $m \geq 0$ additional actual arguments $e_{n+1}, \ldots, e_{n+m}$ after the n required ones $e_1, \ldots, e_n$. The translation combines all arguments, required and additional, into an $(n+m)$-*TUPLE* denoted by square brackets:

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

---

tual loop; the loop body *B* is executed until the Boolean condition *exit* becomes true; since the exit condition is tested before each iteration, the body may not be executed. Body *B* and *init* may be empty.

The correctness of this translation relies on Eiffel's type conformance rule for instances of the *TUPLE* class: every $(n+m)$-*TUPLE* with types $[T_1, \ldots, T_n, T_{n+1}, \ldots, T_{n+m}]$, for $n, m \geq 0$, conforms to any shorter n-*TUPLE* with types $[T_1, \ldots, T_n]$. Thus, translated calls are type-safe because the assignment of actual to formal arguments is covariant [12].

The bodies of variadic functions can refer to the required arguments by name using the usual syntax. The C type *va_list* and functions *va_start* and *va_arg* in library `stdarg` provide a means to access the optional arguments: *va_start*(*argp*, *last*) initializes a variable *argp* of type *va_list* to point to the first optional argument after *last* (the name of the last declared argument). Every successive invocation of *va_arg*(*argp*, *T*) returns the argument of type *T* pointed to by *argp*, and then moves *argp* right after it. The Eiffel translation relies on the helper class *CE_VA_LIST* which replicates `stdarg`'s functionality:

$$
\begin{aligned}
\mathcal{T}_{\text{DE}}(\ va\_list\ argp\ ) \quad &= \quad argp\text{:}\ CE\_VA\_LIST \\
\mathcal{T}(\ va\_start(argp, last)\ ) \quad &= \quad \textbf{create}\ argp.make\ (args, index) \\
\mathcal{T}(\ va\_arg(argp, T)\ ) \quad &= \quad argp.\mathcal{T}_{\text{TY}}(T)\_item
\end{aligned}
$$

where *args* is the name of the argument *TUPLE* in the enclosing variadic function and *index* is the index of the first optional argument.

While *CE_VA_LIST* provides a uniform interface to both Eiffel and C, the actual arguments of a variadic function may also be accessed in the Eiffel translation using the standard syntax for *TUPLE*s: in a variadic function with n required arguments, the expression $args.a_i$ refers to the ith named argument a_i, for $1 \leq i \leq n$; and $args.T_j\_item(j)$ refers to the jth argument a_j, for any $1 \leq j \leq n$ (required argument) or $j > n$ (optional argument), where T_j is a_j's type.

Example 3. Continuing Examples 1 and 2, consider a variadic function *init_cars* that takes a pointer *carsp* to an array of cars and *num* pairs of unsigned integers and strings (passed as optional arguments), and initializes the array with *num* cars whose plate numbers and brands respectively correspond to the integer and string in each pair.

```
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 into Eiffel uses the class *CE_VA_LIST* and accesses the optional unsigned integer arguments with *argp.natural_32_item*, and the optional strings with

argp.pointer_item. The latter returns a generic pointer, which is cast to **char** ∗ with an explicit cast (line 15).

```
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
```

3.4 Unstructured Control Flow

In addition to constructs for structured programming, C offers control-flow breaking
instructions such as jumps. This section discusses their translation to Eiffel.

Jumps. Eiffel enforces structured programming, and hence it lacks control-flow
breaking instructions such as **goto**, **break**, **continue**, and **return**. The translation $\mathcal{T}_{\mathrm{CF}}$
eliminates such instructions along the lines of the *global* version—using Harel's termi-
nology [13]—of the structured programming theorem. The body of a function using
goto determines a list of instructions $s_0, s_1, \ldots, s_n$, where each s_i is a maximal se-
quential block of instructions, with no labels after the first instruction or jumps before
the last one in the block. $\mathcal{T}_{\mathrm{CF}}$ translates the body into a single loop over an auxiliary
integer variable *pc* that emulates a *program counter*:[6]

---

[6]Eiffel's **inspect/when** instructions corresponds to a restricted form of **switch/case** without fall through.

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

Variable pc is initially zero; every iteration of the loop body executes s_{pc} for the current value of pc, and then updates pc (with $upd(pc)$) 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 simpler cases by directly using auxiliary Boolean flag variables with the same names as the instruction they replace. The flags are tested as part of the translated exit conditions for the loops where the control-flow breaking instructions appear. Using this alternative translation scheme where the generality of **goto**s is not required makes for translations with little changes to the code structure, which are usually more readable. The loop **while** $(n > 0)\{$ **if** $(n == 3)$ **break**; $n--$; $\}$, for example, 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**

Long jumps. The C library `setjmp` provides 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 3.5) 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.

Example 4. Consider a C function *target* that executes *setjmp*(*buf*) to save a location using a global variable *buf* of type *jmp_buf*. In the translated Eiffel code, *buf* becomes an attribute of type *CE_EXCEPTION*, and the call to *setjmp* in *target* becomes a creation of an instance attached to *buf*: *buf* := **create** {*CE_EXCEPTION*}.*make* (...); *make*'s actual arguments (omitted for simplicity) contain references to the stack frame of the current instance of *target*, and to the specific instruction where to jump.

Later during the execution, another function *source* called from *target* performs a *longjmp*(*buf*, 3), which diverts execution to the location previously marked by *setjmp* and returns value 3. In Eiffel, the *longjmp* becomes a raising of the exception object *buf* enclosing the return value: *buf.raise* (3).

The flow of what happens next depends on the semantics of Eiffel exceptions, which is significantly different than that of other object-oriented languages such as Java and C#. The raised exception propagates to the recipient *target*, where it is handled by a **rescue** clause. Such exception handling blocks are routine-specific in Eiffel, rather than being associated to arbitrary scopes such as in Java's **try/catch** blocks. The translation of *setjmp* also took care of setting up *target*'s **rescue** clause: if the raised exception stores a reference to the current instance of routine *target*, the code in the **rescue** clause modifies a variable *pc* local to *target* to point to the location of the *setjmp* and, using Eiffel's **retry** instruction, executes *target*'s body again with this new value. Based on the value, the new execution of *target*'s body jumps to the correct location following a mechanism similar to the previously discussed translation of **goto** instructions. Finally, if the handled exception references a routine instance other than the current execution of *target*, the **rescue** clause propagates the exception through the call stack, so that it can reach the correct stack frame.

3.5   Externals and Encapsulation

The translation $\mathcal{T}$ uses classes to wrap header and source files, which give a simple modular structure to C programs.

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` becomes:

> *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 classes translating the other system header files that source.c includes, to have access to their declarations. For example, if foo.c includes stdio.h, *FOO* is declared as **class** *FOO* **inherit** *STDIO*.

3.6 Formatted Output Optimization

The library function *printf* is the standard C output function, which displays values according to format strings passed as argument. In contrast, Eiffel provides the command *Io.put_string* to put plain text strings on standard output, as well as type-specific formatter classes, such as *FORMAT_INTEGER*, to produce string representations of arbitrary types.

With the goal of making the translated code as close as possible to standard Eiffel, C2Eif tries to replace calls to *printf* with equivalent calls to *Io.put_string* and routines of the formatter classes. Whenever *printf*'s returned value is not used and the format string is a constant literal, C2Eif parses the literal format string and encodes it as equivalent calls to *Io.put_string* and formatters. This process may find mismatches between some format specifiers and the types of the corresponding value arguments, which C2Eif reports as warnings. Another situation where C2Eif can replace calls to *printf* with calls to *put_string* is when only one argument is passed to the former, which is then interpreted verbatim as a string. Whenever a warning is issued or the usage of *printf* cannot be rendered using *put_string*, C2Eif falls back to calling a wrapper of the native *printf* function (described in Section 3.5). The translation also implements similar optimizations for variants of *printf* such as *fprintf*.

4 Evaluation and Discussion

This section evaluates the translation $\mathcal{T}$ and its implementation in C2Eif. The bulk of the evaluation assesses correctness (Section 4.1) and performance (Section 4.2) based on experiments with 14 open-source programs. We also qualitatively discuss other aspects that influence maintainability (Section 4.3), the advantages in terms of safety deriving from switching to the Eiffel runtime (Section 4.4), as well as the current limitations of the C2Eif approach (Section 4.5).

4.1 Correct Behavior

To experimentally assess the translations produced by C2Eif, we applied it to 14 open-source C programs, including 7 applications, 6 libraries, and one testsuite; most of them are widely-used in Linux and other "*nix" distributions. `hello world` is the only toy application, which is however useful to have a 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 which 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, a library 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.

Table 1 shows the results of translating the 14 programs into Eiffel using 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 1 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.[7]

We ran extensive trials on the translated programs to verify that they behave as in their original C version, thus validating the correctness of the translation $\mathcal{T}$ and its implementation in C2Eif. The trials comprised informal usage, systematic performance tests for some of the applications, and running the standard testsuites available for the three biggest libraries (`libcurl`, `libgmp`, and `libgsl`). The rest of this section describes the translated testsuites and their behavior with respect to correctness; Section 4.2 discusses the quantitative performance results.

Library `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.

---

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

Table 1: Translation of 14 open-source programs.

| | SIZE (LOCS) | | #EIFFEL | TIME | BINARY SIZE |
	CIL	EIFFEL	CLASSES	(S)	(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

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) which 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.

4.2 Performance

We analyzed the performance of 5 applications, the GCC torture testsuite, and the standard testsuites of 3 libraries (described in Section 4.1) translated to Eiffel using C2Eif. Table 2 shows the result of the performance trials, running on the same system used for the experiments of Section 4.1.[8] For each program or testsuite, Table 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 (columns "C") against the Eiffel translations with C2Eif (columns "T"), and, for the simpler examples, against manually written Eiffel implementations (columns "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 (we manually wrote these transliterations

---

[8]We compiled all CIL-processed C programs with the GCC options `-O2 -s`; and all Eiffel programs with disabled void checking, inlining size 100, and stripped binaries.

Table 2: Performance comparison for 5 applications and 4 testsuites.

	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
`less`	36	36	–	–	–	–	–	–	–
`wget`	16	16	–	22	22	–	4.4	69	–
`vim`	85	85	–	–	–	–	–	–	–
`libcurl`	199	212	–	–	–	–	–	–	–
`libgmp`	44	728	–	–	–	–	–	–	–
`libgsl`	25	1501	–	–	–	–	–	–	–
`gcc` (torture)	0	5	–	–	–	–	–	–	–

ourselves). Maximum CPU and RAM usages are immaterial for the testsuites (torture and libraries), because their execution consists of a large number of separate calls. The rest of this section discusses the performance results together with other qualitative performance assessments.

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 is faster than the manually written Eiffel transliteration. 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, and 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, thus the maximum memory usage is higher in operational conditions).

The test with `libcurl` consisted in running all 583 tests from the standard test-suite 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 slowlier 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.

The interactive applications (`xeyes`, `less`, `links`, and `vim`) run smoothly with good responsiveness, comparable to their original implementations. The test for `less` and `vim` consisted in scrolling through large text files one line at a time (by holding "arrow down" in `less`) or one page at a time (by holding "page down" in `vim`). Table 2 shows the running times, which are the same in C and Eiffel; the screen refresh rate is also indistinguishable.

In all, the performance overhead in switching from C to Eiffel significantly varies with the program type but, even when it is significant, 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.

4.3   Usability and 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. To give an idea of how C2Eif fares in this respect, consider writing a simple class *PRINT_SOURCE_SEHOME*, which uses the API of cURL to retrieve and print the HTML code of the home page at http://se.inf.ethz.ch. Table 3 shows two versions of this client class. The one on the left uses the translation of `libcurl` automatically generated by C2Eif; the one on the right uses the wrapper of `libcurl` part of the Eiffel standard libraries included with the EiffelStudio IDE, written by EiffelSoftware programmers. The two

solutions are quite similar in structure and style. The version based on C2Eif still requires adhering to a couple of conventions that are a legacy of the translation from C: the arguments passed to routine *curl_easy_setopt* must be listed in a tuple, and the routine call itself includes a *do_nothing* which is needed whenever a function that returns a value is used as an instruction (Eiffel enforces separation between functions and procedures). On the other hand, the "native" solution has a slightly more complex control structure, because it has to check that the dynamically linked cURL library is actually accessible at runtime; this is unnecessary with the implementation based on C2Eif, which does not depend on dynamically linked libraries since `libcurl` is available translated to Eiffel.

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

```
class                                          class
    PRINT_SOURCE_SEHOME                            PRINT_SOURCE_SEHOME

inherit                                        inherit
    LIBCURL_CONSTANTS                              CURL_OPT_CONSTANTS

create                                         create
    make                                           make

feature                                        feature

    make                                           make
        local                                         local
            easy: P_EASY_STATIC                            easy: CURL_EASY_EXTERNALS
            handle: CE_POINTER [ANY]                       handle: POINTER
            ret: NATURAL_32                                ret: INTEGER_32
        do                                             do
            create easy                                    create easy
            handle := easy.curl_easy_init                  if easy.is_dynamic_library_exists then
            easy.curl_easy_setopt ([handle, Curlopt_url,       handle := easy.init
              ce_string ("se.inf.ethz.ch")]).do_nothing      easy.setopt_string (handle, Curlopt_url,
            ret := easy.curl_easy_perform (handle)             "se.inf.ethz.ch")
            easy.curl_easy_cleanup (handle)                  ret := easy.perform (handle)
        end                                                  easy.cleanup (handle)
end                                                    else
                                                           Io.error.put_string ("cURL not found.%N")
                                                       end
                                                   end
                                           end
```

Table 3: Two Eiffel clients of cURL: using `libcurl` translated with C2Eif (left) and using the library wrapper provided by EiffelStudio (right).

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 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 briefly discuss in Section 6, 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.

4.4 Safety and Debuggability

Besides the obvious advantage of reusing the huge C code-base, translating C code to Eiffel using C2Eif leverages some high-level feature which may improve *safety* and make *debugging* easier in some conditions.

Uncontrolled format string is a well-known vulnerability [4] 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 would be the instruction *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 usages of *printf* 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 1. C2Eif translates more complex usages 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, and 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, and 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's 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, and 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 usages 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 3.1), 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 has been included in more recent versions of the library); 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 following 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 multidimensional 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.

4.5 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 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.

Another limitation is the fact that C2Eif takes input from CIL, and hence it does not support legacy C such as K&R C. The support can, however, be implemented by di-

25

rectly extending the pre-processing CIL front-end. Similarly, the GCC torture testsuite highlighted a few exotic GCC features currently unsupported by C2Eif (Section 4.1), which may be handled in future work.

Code using unrestricted **goto**s poses the biggest hurdles to producing readable Eiffel code. This is arguably unavoidable when translating to any programming language that does not have jumps. The translation $\mathcal{T}$ does, however, avoid the most complicated general translation scheme with the simpler control-flow breaking instructions **continue**, **break**, and **return** instructions, whose translation is normally much more readable than when using unrestricted **goto**s (see the example in Section 4).

5   Related Work

There are two main approaches to reuse source code written in a "foreign" language (e.g., C) in a different "host" language (e.g., Eiffel): define wrappers for the components written in the foreign language; and translate the foreign source code into functionally equivalent host code. We discuss related work pursuing these approaches in Sections 5.1 and 5.2. In Section 5.3, we review the major solutions to automate object-oriented reengineering, which is a natural follow-up of automatic translation into object-oriented languages.

5.1   Wrapping Foreign Code

Wrappers enable the reuse of foreign implementations through the API of bridge libraries. This approach (e.g., [6, 5, 30]) does not modify the foreign code, whose functionality is therefore not altered; moreover, the complete foreign language is supported. On the other hand, the type of data that can be retrieved through the bridging API is often restricted to a simple subset common to the host and foreign language (e.g., primitive types). C2Eif uses wrappers only to translate external functions and assembly code.

5.2   Translating Foreign Code

Industrial practices have long encompassed manual migrations of legacy code. Some semi-automated tools exist that help translate code written in legacy programming languages such as old versions of COBOL [37, 21], Fortran-77 [1, 34], and K&R C [46].

Terekhov et al. [37] review how automated language conversion is applied in industry; based on their experience, they conclude that "creating 100% automated conversion tools is neither possible, nor desirable". Our experience with C2Eif, however, suggests that such a conclusion has only relative validity: in the rich design space of automatic translators, there are scenarios where trading off some performance for complete automation is possible and desirable. Rather than imposing an upfront heavy-weight burden on developers in charge of migration, we suggest to start with an automatically translated version which is suboptimal but works out of the box, and then devote the manual programming effort to improving and adapting what is necessary—

incrementally with an agile approach which also depends on the specific application domain and requirements.

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 [25, 32, 45], Ada to Ada95 [35], and C to C++ [15, 47]. Some of such efforts try to 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 (see the focused discussion in Section 5.3). C2Eif follows a similar approach, but also takes advantage of some advanced features (such as contracts) to improve the reliability of translated code.

Ephedra [18] 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 cannot translate frequently used features such as unrestricted **goto**s, external pre-compiled libraries, and inlined assembly code. A case study evaluating Ephedra [19] 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. In contrast, C2Eif is completely automatic, and works with significantly larger programs.

Other tools (proprietary or open-source) to translate C (and C++) to Java or C# include: C2J++ [39], C2J [27], and C++2C# and C++2Java [36]. Table 4 shows a feature comparison among the currently available tools that translate C to an object-oriented language, 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 a couple of cases we could only find papers describing the tool but not a version of the implementation working on standard machines.

- An (subjective to a certain extent) assessment of the *readability* of the code produced. In each case, we tried to evaluate 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's readability negatively because the tool puts data into a single global array to support pointer arithmetic. This is quite detrimental to readability and also circumvents type checking in the Java translation.

- Whether the tool supports unrestricted calls to *external libraries*, unrestricted *pointer arithmetic*, unrestricted **goto**s, and inlined *assembly code*.

The table demonstrates that C2Eif is arguably the first completely automatic tool that handles the complete C language as used in practice.

In previous work, we developed J2Eif, an automatic source-to-source translator from Java to Eiffel [43]; translating between two object-oriented languages does not

Table 4: Tools translating C to O-O languages.

	target language	completely automatic	available	readability	external libraries	pointer arithmetic	**gotos**	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

pose some of the formidable problems of bridging wildly different abstraction levels, which C2Eif had to deal with.

TXL [2] is an expressive programming language designed to support source code analysis and transformation. It has been used to implement several language translation frameworks such as from Java to TCL and to Python. We could have used TXL to implement C2Eif; using regular Eiffel, however, allowed us to be independent of third-party closed-source tools, and to retain complete control over the implemented functionalities.

Safer C. Many techniques exist aimed at ameliorating the safety of existing C code; for example, detection of format string vulnerability [3], out-of-bound array accesses and other memory errors [28, 24], or type errors [23]. 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 [7], which also finds errors in C program as a "side effect" of a rigorous translation.

5.3 Object-Oriented Reengineering

After code written in a procedural language has been migrated to an object-oriented environment, it is natural to reengineer it to conform to the object-oriented design style, taking full advantage of features such as inheritance; this is the goal of object-oriented reengineering. Object-oriented reengineering is beyond the scope of this paper; and we tackled it in follow-up work based on C2Eif which we mention in Section 6. Nonetheless, it is still useful to compare existing approaches to reengineering with C2Eif, solely based on features such as degree of automation and tool support; see our follow-up work [41] for a discussion focused on the object-oriented reengineering techniques.

Table 5 summarizes some features of the main approaches to object-oriented reengineering of procedural code:

- The *source* and the *target* languages (or if it is a generic methodology).

28

Table 5: Comparison of approaches to O-O reengineering.

	source–target	tool support	completely automatic	full language	evaluated
Gall [11]	methodology	no	no	–	yes
Jacobson [14]	methodology	no	no	–	yes
Livadas [17]	C–C++	yes	no	no	no
Kontogiannis [16]	C–C++	yes	no	?	10KL
Frakes [10]	C–C++	yes	no	no	2KL
Fanta [8]	C++–C++	yes	no	no	120KL
Newcomb [26]	Cobol–OOSM	yes	yes	no	168KL
Mossienko [21]	Cobol–Java	yes	no	no	25KL
Sneed [33]	Cobol–Java	yes	yes	no	200KL
Sneed [31]	PL/I–Java	yes	yes	no	10KL
C2Eif	C–Eiffel	YES	yes	yes	932KL

- Whether *tool support* was developed, that is whether there exists a tool or the paper explicitly mentions the implementation of a tool. A YES in small caps denotes the only currently publicly available tool, namely C2Eif.

- Whether the approach is *completely automatic*, that is does not require any user input other than providing a source procedural program.

- Whether the approach supports the *full* source *language* (as used in practice) or only a subset thereof.

- Whether the approach has been *evaluated*, that is whether the paper mentions evidence, such as a case study, that the approach was tried on real programs. If available, the table indicates the size of the programs used in the evaluation.

Newcomb's [26] and Sneed's [33] are the only automatic tools which have been evaluated on programs of significant size. Newcomb's tool [26], however, produces hierarchical object-oriented state machine models (OOSM); the mapping from OOSM to an object-oriented language is out of the scope of the work. Sneed's tool [33] translates Cobol to Java; the paper reports that manual corrections of the automatically generated code were needed to get to a correct translation. While these corrections have successively been incorporated as extensions of the tool, the full Cobol language remains unsupported, according to the paper.

6 Conclusions and Future Work

This paper presented the complete translation of C applications into Eiffel, and its implementation into the freely available automatic tool C2Eif. C2Eif supports the com-

plete C language, including unrestricted pointer arithmetic and pre-compiled libraries. Experiments in the paper showed that C2Eif correctly translates complete applications and libraries of significant size, and takes advantage of some of Eiffel's advanced features to produce code of good quality.

Future work. Future work will improve the readability and maintainability of the generated code. CIL, in particular, optimizes the code for program analysis, which is sometimes detrimental to readability of the Eiffel code generated by C2Eif. For example, CIL does not preserve comments, which are therefore lost in translation. We will also optimize the helper classes to improve on the few performance bottlenecks mentioned in Section 4.2.

A major follow-up to the work described in this paper is the object-oriented reengineering of C code translated to Eiffel. In recent work [41], we have developed an automatic reengineering technique, and implemented it atop the translation produced by C2Eif and described in the present paper. The technique encapsulates functions and type definitions into classes that achieve low coupling and high cohesion, and introduces inheritance and contracts when possible. Other future work still remains in the direction of reengineering, such as in automatically replacing C data structure implementations (e.g., hash tables) with their Eiffel equivalents.

References

[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] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

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

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

[5] Andrea de Lucia, Giuseppe A. Di Lucca, Anna Rita Fasolino, Patrizia Guerra, and Silvia Petruzzelli. Migrating legacy systems towards object-oriented platforms. *Proc. of ICSM*, pages 122–129, 1997.

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

[7] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

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

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

[10] William Frakes, Gregory Kulczycki, and Natasha Moodliar. An empirical comparison of methods for reengineering procedural software systems to object-oriented systems. In *ICSR*, volume 5030 of *LNCS*, pages 376–389, 2008.

[11] H. Gall and R. Klosch. Finding objects in procedural programs: an alternative approach. In *WCRE*, pages 208–216. IEEE, 1995.

[12] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. Wiley, 3rd edition, 1998.

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

[14] Ivar Jacobson and Fredrik Lindström. Reengineering of old systems to an object-oriented architecture. In *OOPSLA*, pages 340–350. ACM, 1991.

[15] Kostas Kontogiannis and Prashant Patil. Evidence driven object identification in procedural code. In *STEP*, pages 12–21, 1999.

[16] Kostas Kontogiannis and Prashant Patil. Evidence driven object identification in procedural code. In *STEP*, pages 12–21. IEEE, 1999.

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

[18] Johannes Martin and Hausi A. Müller. Strategies for migration from C to Java. In *CSMR*, pages 200–210. IEEE Computer Society, 2001.

[19] Johannes Martin and Hausi A. Müller. C to Java migration experiences. In *CSMR*, pages 143–153. IEEE Computer Society, 2002.

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

[21] M. Mossienko. Automated Cobol to Java recycling. In *CSMR*, pages 40–50. IEEE, 2003.

[22] 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 Compilier Construction*, pages 213–228, 2002.

[23] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL*, pages 128–139, 2002.

[24] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007.

[25] P. Newcomb and G. Kotik. Reengineering procedural into object-oriented systems. In *WCRE*, pages 237–249, 1995.

[26] P. Newcomb and G. Kotik. Reengineering procedural into object-oriented systems. In *WCRE*, pages 237–249. IEEE, 1995.

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

[28] 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.

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

[30] Miguel A. Serrano, Doris L. Carver, and Carlos Montes de Oca. Reengineering legacy systems for distributed environments. *J. Syst. Softw.*, 64(1):37–55, 2002.

[31] H. Sneed. Migrating PL/I code to Java. In *CSMR*, pages 287–296. IEEE, 2011.

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

[33] H.M. Sneed. Migrating from COBOL to Java. In *ICSM*, pages 1–7. IEEE, 2010.

[34] Gokul V. Subramaniam and Eric J. Byrne. Deriving an object model from legacy Fortran code. *ICSM*, pages 3–12, 1996.

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

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

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

[38] The Language Popularity Index. http://lang-index.sourceforge.net, 2011.

[39] Eli Tilevich. Translating C++ to Java. In *First German Java Developers' Conference Journal, Sun Microsystems Press*. IEEE Computer Society, 1997.

[40] Marco Trudel, Carlo A. Furia, and Martin Nordio. Automatic C to O-O translation with C2Eiffel. In Rocco Oliveto, Denys Poshyvanyk, James Cordy, and Thomas Dean, editors, *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*, pages 508–509. IEEE Computer Society, October 2012. Tool demonstration paper.

[41] Marco Trudel, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Really automatic scalable object-oriented reengineering. In Giuseppe Castagna, editor, *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lecture Notes in Computer Science*, pages 477–501. Springer, July 2013.

[42] Marco Trudel, Carlo A. Furia, Martin Nordio, Bertrand Meyer, and Manuel Oriol. C to O-O translation: Beyond the easy stuff. In Rocco Oliveto, Denys Poshyvanyk, James Cordy, and Thomas Dean, editors, *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*, pages 19–28. IEEE Computer Society, October 2012.

[43] Marco Trudel, Manuel Oriol, Carlo A. Furia, and Martin Nordio. Automated translation of Java source code to Eiffel. In *TOOLS Europe*, volume 6705 of *LNCS*, pages 20–35, 2011.

[44] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, volume 7041 of *LNCS*, pages 382–398, 2011.

[45] Theo Wiggerts, Hans Bosma, and Erwin Fielt. Scenarios for the identification of objects in legacy systems. In *WCRE*, pages 24–32, 1997.

[46] A. Yeh, D. Harris, and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *WCRE*, pages 227–236, 1995.

[47] Ying Zou and Kostas Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *APSEC*, pages 390–399, 2001.