C to O-O Translation: Beyond the Easy Stuff

Marco Trudel*, Carlo A. Furia*, Martin Nordio*, Bertrand Meyer* and Manuel Oriol†
* Chair of Software Engineering, ETH Zurich, Switzerland
Email: firstname.lastname@inf.ethz.ch
† Dept. of Computer Science, University of York, York, UK; manuel@cs.york.ac.uk
ABB Corporate Research, Industrial Software Systems, Baden-Dättwil, Switzerland; manuel.oriol@ch.abb.com

***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 supports 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.**

I. Introduction

Programming languages have significantly evolved since the original design of C in the 1970's as a "system implementation language" [1] 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 [2], 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 [3], and countless C applications are still 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 applications into Eiffel, an object-oriented programming language, and its implementation C2Eif. While the most common approaches to re-use C code in other host languages are based on "foreign function APIs" (see Section V for examples), 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 review in Section V). A limitation of the resulting tools is that they hardly handle the trickier or specialized parts of the C language [4], 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 IV, including major utilities such as the `vim` editor (276 KLOC), major libraries such as `libgsl` (238 KLOC), and the challenging GCC "torture" tests for C compilers.

C2Eif is open source and available for download[1]:
http://se.inf.ethz.ch/research/c2eif

Sections II–III 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 [5]; 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[2] Eiffel code.*

Safer code. Translating C code to Eiffel with C2Eif is quite useful to reuse C applications in a modern environment, but it also implies several valuable side-benefits—demonstrated in Section IV. First, the translated code blends well with hand-written Eiffel code because it is not a mere transliteration

[1] The webpage includes C2Eif's sources, pre-compiled binaries, source and binaries of all translated programs in Table I, and a user guide.

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

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 IV-C 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 applications 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.

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 [6] through the native support of contracts. In addition, 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 likely applicable with little effort to other object-oriented languages such as Java and C#; we plan to pursue this direction in future work.

An extended version of this paper, including a few more examples and details, is available as technical report [7]. A companion paper presents the tool C2Eif from the user's point of view [8].

II. Overview and Architecture

C2Eif is a compiler with graphical user interface that translates C programs to Eiffel programs. The translation is a complete Eiffel application that replicates the functionalities of the C source application. C2Eif is implemented in Eiffel.

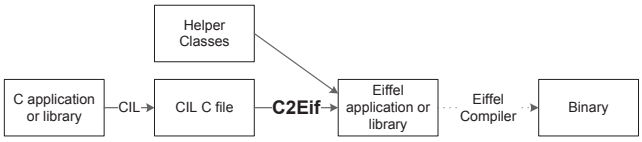

Fig. 1. Overview of how C2Eif works.

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 [5] 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 40 such transformations (i.e., $n = 40$). 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 those of 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 III 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 translating **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 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.

III. 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 II) but occasionally overlook inessential details for greater 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 II) includes an attribute *c_pointer* of type *POINTER* that offers access to a C-conforming representation usable in **external** calls.

A. 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 to Eiffel using class *CE_POINTER*[*G*] with the generic parameter *G* instantiated with the pointed type:

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

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*:

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

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 III-B 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*[$\mathcal{T}_{\mathrm{TY}}(T)$]. The size parameter n, if present, does not affect the declaration, but initializations of array variables use it (see Section III-B). 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 **enum** type *E* defined or used, the translation introduces an Eiffel class *E* defined by the translation $\mathcal{T}_{\mathrm{TD}}$ (for type definition):

$\mathcal{T}_{\mathrm{TD}}$(**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, $\mathcal{T}_{\mathrm{TY}}$(**enum** *E*) = *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 $\mathcal{T}_{\mathrm{TD}}$(**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

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

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

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 descendant, 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 $\mathcal{T}_{\mathrm{TD}}$ 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
```

B. 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*(≪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 IV-C). 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*(**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 *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'** 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*.

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`.

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).

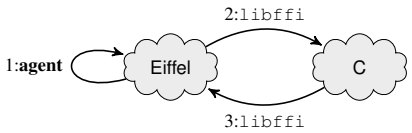

Fig. 2. Function pointers.

The services of class *CR_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.

C. Control Flow

This section discusses the translation $\mathcal{T}_{\mathrm{CF}}$ of instructions directing the control flow. **Sequential** composition, **conditionals**, and **loops** are quite similar in C and Eiffel, hence have a straightforward translation (see [7]).

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 [9]—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 without 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*:

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

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.

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

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

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:

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

We can access arguments in a *TUPLE* either with standard Eiffel syntax, or using the helper class *CE_VA_LIST*. With standard syntax, $args.a_i$ refers to the required argument with name $a_i, 1 \leq i \leq n$, whereas $args.\mathcal{T}_{\mathrm{TY}}(T_k)\_item(k)$ 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)
e1 := va_list.T_TY(T_{n+1})_item -- first optional argument a_{n+1}
e2 := va_list.T_TY(T_{n+2})_item -- second optional argument a_{n+2}
...
```

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 III-D) 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.

D. 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 **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`. Each class also *inherits* from classes created from external functions corresponding to included system headers, to access their features. For example, if `foo.c` includes `stdio.h`, *FOO* is declared as **class** *FOO* **inherit** *STDIO*.

IV. Evaluation

This section evaluates the translation $\mathcal{T}$ and its implementation in C2Eif with 10 programs and 4 testsuites.

A. Correct Behavior

Table I shows data about 10 C open-source programs and a testsuite translated to 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 7.0.8. For each application, library, and testsuite Table I 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.[5]

The 10 programs include 7 applications and 3 libraries; all of them are widely-used in Linux and other "*nix" distributions. `hello world` is the only toy application, which is however useful as 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: `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 usage, we performed systematic performance tests for some of the applications (described below), and ran standard testsuites on the 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. 1 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

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

	Size (LOCS)		#Eiffel	Translation	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
`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	898,037	1,334,168	5,433	413	1,626

TABLE I
Translation of 10 open-source programs and a testsuite.

a copy-paste error rather than a feature. 2 tests exercise GCC-specific optimizations, which are immaterial after translation to Eiffel. 6 tests target somehow exotic GCC built-in functions, such as *builtin_frame_address*; 1 test performs explicit function alignment; and 3 rely on special bitfield operations.

B. Performance

Table II shows the result of trials that analyze the performance of 6 of the programs, plus the GCC torture testsuite, running on the same system as Table I. For each program or testsuite, Table II 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).

C. 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 [10] of C's *printf* library function, which permits malicious

	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 II
PERFORMANCE COMPARISON FOR 3 OPEN-SOURCE APPLICATIONS AND 4 TESTSUITES.

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 `./a.out "{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 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 I. 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, 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 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 III-A), 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.

D. 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 IV-C 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, hence it does not support legacy C such as K&R C. The support can, however, be implemented by directly extending the pre-processing CIL front-end. Similarly, the GCC torture testsuite highlighted a few exotic GCC features currently unsupported by C2Eif (Section IV-B), which may be handled in future work.

V. 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.

Wrapping foreign code. Wrappers enable the reuse of foreign implementations through the API of bridge libraries. This approach (e.g., [11], [12], [13]) 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.

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 [14], [15], Fortran-77 [16], [17], and K&R C [18].

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 [19], [20], [21], Ada to Ada95 [22], and C to C++ [23], [24]. 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. C2Eif follows a similar approach, but it also takes advantage of some advanced features (such as contracts) to improve the reliability of translated code.

Ephedra [25] 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 cannot translate frequently used features such as unrestricted **goto**s, 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. By contrast, C2Eif is completely automatic, and it works with significantly larger programs.

Other tools (proprietary or open-source) to translate C (and C++) to Java or C# include: C2J++ [27], C2J [28], and C++2C# and C++2Java [29]. Table III 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 and produces readable object-oriented code.

In previous work, we developed J2Eif, an automatic source-to-source translator from Java to Eiffel [30]; translating between two object-oriented languages does not pose some of the

	target language	completely automatic	available	readability	external libraries	pointer arithmetic	**goto**s	inlined assembly
`C2Eif`	Eiffel	yes	yes	+	yes	yes	yes	yes
`Ephedra`	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 III
TOOLS TRANSLATING C TO O-O LANGUAGES.

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

Safer C. Many techniques exist aimed at ameliorating the safety of existing C code; for example, detection of format string vulnerability [31], out-of-bound array accesses and other memory errors [32], [33], or type errors [34]. 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 [4], which also finds errors in C program as a "side effect" of a rigorous translation.

VI. CONCLUSIONS AND FUTURE WORK

This paper presented the complete translation of C applications into Eiffel, and its implementation into the freely available tool C2Eif. 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 that is safer.

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. Another aspect to be improved is the object-oriented re-engineering of C code translated to Eiffel. We will investigate more sophisticated encapsulation of globals into classes, usage of inheritance, re-factoring routine arguments into class attributes, command/query separation practices [2], and replacing C data structure implementations (e.g., hash tables) with their Eiffel equivalents.

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

REFERENCES

[1] D. Ritchie, "The development of the C language," in *History of Programming Languages Conference (HOPL-II) Preprints*, 1993, pp. 201–208.
[2] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.
[3] The Language Popularity Index, http://lang-index.sourceforge.net, 2011.
[4] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," in *POPL*, 2012, pp. 533–544.
[5] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs," in *Conference on Compilier Construction*, 2002, pp. 213–228.
[6] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer, "Usable verification of object-oriented programs by combining static and dynamic techniques," in *SEFM*, ser. LNCS, vol. 7041, 2011, pp. 382–398.
[7] M. Trudel, C. A. Furia, M. Nordio, B. Meyer, and M. Oriol, "Automatic translation of C source code to Eiffel," http://arxiv.org/abs/1206.5648, June 2012.
[8] M. Trudel, C. A. Furia, and M. Nordio, "Automatic C to O-O translation with C2Eiffel," in *WCRE*, 2012, tool demonstration paper.
[9] D. Harel, "On folk theorems," *Commun. ACM*, vol. 23, no. 7, pp. 379–389, 1980.
[10] CWE-134, "Uncontrolled format string," http://cwe.mitre.org/data/definitions/134.html.
[11] W. C. Dietrich, Jr., L. R. Nackman, and F. Gracer, "Saving legacy with objects," *SIGPLAN Not.*, vol. 24, no. 10, pp. 77–83, 1989.
[12] A. de Lucia, G. A. D. Lucca, A. R. Fasolino, P. Guerra, and S. Petruzzelli, "Migrating legacy systems towards object-oriented platforms," *Proc. of ICSM*, pp. 122–129, 1997.
[13] M. A. Serrano, D. L. Carver, and C. M. de Oca, "Reengineering legacy systems for distributed environments," *J. Syst. Softw.*, vol. 64, no. 1, pp. 37–55, 2002.
[14] A. A. Terekhov and C. Verhoef, "The realities of language conversions," *IEEE Software*, vol. 17, no. 6, pp. 111–124, 2000.
[15] M. Mossienko, "Automated Cobol to Java recycling," in *CSMR*. IEEE, 2003, pp. 40–50.
[16] B. L. Achee and D. L. Carver, "Creating object-oriented designs from legacy FORTRAN code," *Journal of Systems and Software*, vol. 39, no. 2, pp. 179–194, 1997.
[17] G. V. Subramaniam and E. J. Byrne, "Deriving an object model from legacy Fortran code," *ICSM*, pp. 3–12, 1996.
[18] A. Yeh, D. Harris, and H. Reubenstein, "Recovering abstract data types and object instances from a conventional procedural language," in *WCRE*, 1995, pp. 227–236.
[19] P. Newcomb and G. Kotik, "Reengineering procedural into object-oriented systems," in *WCRE*, 1995, pp. 237–249.
[20] H. Sneed, "Migration of procedurally oriented Cobol programs in an object-oriented architecture," in *Software Maintenance*, 1992, pp. 105–116.
[21] T. Wiggerts, H. Bosma, and E. Fielt, "Scenarios for the identification of objects in legacy systems," in *WCRE*, 1997, pp. 24–32.
[22] R. Sward, "Extracting ada 95 objects from legacy ada programs," in *Reliable Software Technologies - Ada-Europe 2004*, ser. Lecture Notes in Computer Science, A. Llamosí and A. Strohmeier, Eds., vol. 3063. Springer, 2004, pp. 65–77.
[23] K. Kontogiannis and P. Patil, "Evidence driven object identification in procedural code," in *STEP*, 1999, pp. 12–21.
[24] Y. Zou and K. Kontogiannis, "A framework for migrating procedural code to object-oriented platforms," in *APSEC*, 2001, pp. 390–399.
[25] J. Martin and H. A. Müller, "Strategies for migration from C to Java," in *CSMR*. IEEE Computer Society, 2001, pp. 200–210.
[26] ——, "C to Java migration experiences," in *CSMR*. IEEE Computer Society, 2002, pp. 143–153.
[27] E. Tilevich, "Translating C++ to Java," in *First German Java Developers' Conference Journal, Sun Microsystems Press*. IEEE Computer Society, 1997.
[28] Novosoft, "C2J: a C to Java translator," http://www.novosoft-us.com/solutions/product_c2j.shtml, 2001.
[29] Tangible Software Solutions, "C++ to C# and C++ to Java," http://www.tangiblesoftwaresolutions.com/.
[30] M. Trudel, M. Oriol, C. A. Furia, and M. Nordio, "Automated translation of Java source code to Eiffel," in *TOOLS Europe*, ser. LNCS, vol. 6705, 2011, pp. 20–35.
[31] C. Cowan, "FormatGuard: Automatic protection from `printf` format string vulnerabilities," in *Proceedings of the 10th USENIX Security Symposium*, 2001.
[32] Y. Oiwa, "Implementation of the memory-safe full ANSI-C compiler," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 259–269.
[33] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007, pp. 89–100.
[34] G. C. Necula, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy code," in *POPL*, 2002, pp. 128–139.