Automated Translation of Java Source Code to Eiffel

Marco Trudel[1], Manuel Oriol[2], Carlo A. Furia[1], and Martin Nordio[1]

[1] Chair of Software Engineering, ETH Zurich, Switzerland
{marco.trudel, carlo.furia, martin.nordio}@inf.ethz.ch
[2] University of York, United Kingdom
{manuel@cs.york.ac.uk}

Abstract. Reusability is an important software engineering concept actively advocated for the last forty years. While reusability has been addressed for systems implemented using the same programming language, it does not usually handle interoperability with different programming languages. This paper presents a solution for the reuse of Java code within Eiffel programs based on a source-to-source translation from Java to Eiffel. The paper focuses on the critical aspects of the translation and illustrates them by formal means. The translation is implemented in the freely available tool J2Eif; it provides Eiffel replacements for the components of the Java runtime environment, including Java Native Interface services and reflection mechanisms. Our experiments demonstrate the practical usability of the translation scheme and its implementation, and record the performance slow-down compared to custom-made Eiffel applications: automatic translations of *java.util* data structures, *java.io* services, and SWT applications can be re-used as Eiffel programs, with the same functionalities as their original Java implementations.

1 Introduction

Code reuse has been actively advocated for the past forty years [12], has become a cornerstone principle of software engineering, and has bred the development of serviceable mechanisms such as modules, libraries, objects, and components. These mechanisms are typically language-specific: they make code reuse practical within the boundaries of the same language, but the reuse of "foreign" code written in a specific language within a program written in a different "host" language is a problem still lacking universally satisfactory solutions. The reuse of foreign code is especially valuable for languages with a small development community: some programmers may prefer the "host" language because its design and approach are more suitable for their application domain, but if only a small community uses this language, they also have to wait for reliable implementations of new services and libraries unless there is a way to reuse the products available, sooner and in better form, for a more popular "foreign" language. For example, the first Eiffel library offering encryption[3] was released in 2008 and still

[3] http://code.google.com/p/eiffel-encryption-library/

is in alpha status, while Java has offered encryption services in the *java.security* standard package since 2002.

A straightforward approach to reuse foreign code is to wrap it into components and access it natively through a bridge library which provides the necessary binding. This solution is available, for example, in Eiffel to call external C/C++ code—with the *C-Eiffel Call-In* Library (CECIL)—and Java code—with the *Eiffel2Java* Library; the Scala language achieves interoperability with Java using similar mechanisms. Such bridged solutions execute the foreign code in its native environment which is not under direct control of the host's; this introduces potential vulnerabilities as guarantees of the host environment (provided, for example, by its static type system) may be violated by the uncontrolled foreign component. More practically, controlling the foreign components through the interface provided by the bridge is often cumbersome and results in code difficult to maintain. For example, creating an object wrapping an instance of *java.util.LinkedList* and adding an element to it requires six instructions with Eiffel2Java, some mentioning Java API's signatures encoded as strings such as *method_id* := *list.method_id* (**"add"**, **"(Ljava/lang/Object;)Z"**).

A source-to-source translation of the foreign code into the host does not incur the problems of the bridged solutions because it builds a functionally equivalent implementation in another language. The present paper describes a translation of Java source into Eiffel and its implementation in the tool J2Eif [8]. While Eiffel and Java are both object-oriented languages, the translation of one into the other is tricky because superficially similar constructs, such as those for exception handling, often have very different semantics. In fact, correctness is arguably the main challenge of source-to-source translation: Section 3 formalizes the most delicate aspects of the translation to describe how they have been tackled and to give confidence in the correctness of the translation.

As shown by experiments in Section 4, J2Eif can translate non-trivial Java applications into functionally equivalent Eiffel ones; the system also provides replicas of Java's runtime environment and a precompiled JDK standard library. The usage of the translated code is, in most cases, straightforward for Eiffel programmers; for example, creating an instance l of *java.util.LinkedList* and adding an element e to it becomes the mundane (at least for Eiffel programmers):

create *l.make_JAVA_UTIL_LINKEDLIST* ; $r :=$ *l.method_add_from_object* (e)

Since Eiffel compiles to native code, a valuable by-product of J2Eif is the possibility of compiling Java applications to native code. The experiments in Section 4 show that Java applications automatically translated into Eiffel with J2Eif incur in a noticeable slow-down—especially those making an intense use of translated data-structure implementations. The slow-down is unsurprising, as a generic, automated translation scheme is no substitute for a carefully designed re-engineering that makes use of Eiffel's peculiarities. Using J2Eif, however, enables the fast reuse of new Java libraries in Eiffel applications—a valuable service to access Java's huge codebase in a form congenial to Eiffel programmers. Performance enhancement belongs to future work.

Section 2 gives an overview of the architecture of J2Eif; Section 3 describes the translation in detail; Section 4 evaluates the implementation with four experiments and points out its major limitations; Section 5 discusses related work; Section 6 concludes.

2 Design Principles

J2Eif [8] is a stand-alone compiler with graphical user interface that translates Java programs to Eiffel programs. The translation is a complete Eiffel application which replicates the functionalities of the Java source application by including replacements of the Java runtime environment (most notably, the Java Native Interface and reflection mechanisms). J2Eif is implemented in Java.

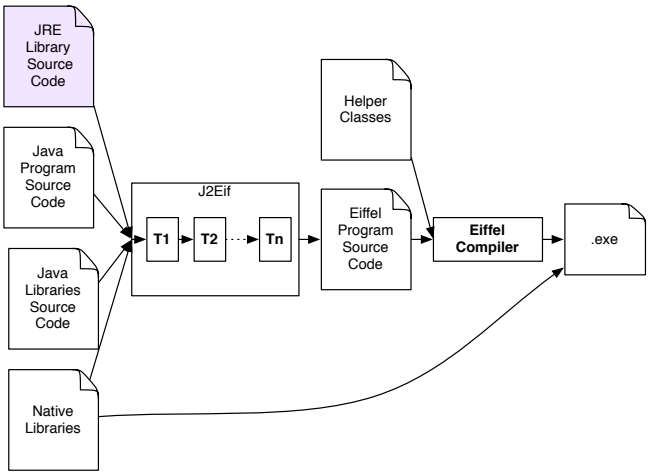

Fig. 1. High-level view of J2Eif.

High-level view. Figure 1 shows the high-level usage of J2Eif. To translate a Java program, the user provides the source code of the program, its Java dependencies, as well as any external native libraries referenced by the program. J2Eif produces Eiffel source code that can be compiled by an Eiffel compiler such as EiffelStudio. Native libraries called by native methods in Java are then directly called from Eiffel. While J2Eif can compile the complete code of the Java Runtime Environment (JRE) library source, it comes with a precompiled version which drastically reduces the overall compilation time.

Translation. J2Eif implements the mapping $\mathcal{T}$: Java $\rightarrow$ Eiffel of Java code into Eiffel code. Both languages follow the object-oriented paradigm and hence share

several notions such as objects, classes, methods, and exceptions. Nonetheless, the semantics of the same notion in the two languages are often quite different. Section 3 describes all the aspects taken into account by the translation and focuses on its particularly delicate features by formalizing them.

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

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

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

3 Translating Java to Eiffel

This section describes the salient features of the translation $\mathcal{T}$ from Java to Eiffel, grouped by topic. Eiffel and Java often use different names for comparable object-oriented concepts; to avoid ambiguities, the present paper matches the terms in the presentation, whenever possible without affecting readability, and uses only the appropriate one when discussing language-specific aspects. Table 1 lists the Java and Eiffel names of fundamental object-oriented concepts.

Java	**Eiffel**
class	class
abstract/interface	deferred
concrete	effective
exception	exception

Java	**Eiffel**
member	feature
field	attribute
method	routine
constructor	creation procedure

Table 1. Object-oriented terminology in Java and Eiffel.

3.1 Language Features

We formalize some components of $\mathcal{T}$ by breaking it down into simpler functions denoted by ∇; these functions are a convenient way to formalize $\mathcal{T}$ and, in

general, different than the transformations T_i discussed in Section 2; the end of the present section sketches an example of differences between ∇'s and T_i's. The following presentation ignores the renaming scheme, discussed separately (Section 3.4), and occasionally overlooks inessential syntactic details. The syntax of Eiffel's exception handling adheres to the working draft 20.1 of the ECMA Standard 367; adapting it to work with the syntax currently supported is trivial.

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

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

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

JAVA_PARENT is ancestor to every translated class, to which it provides helper routines for various services such as access to the native interface, exceptions, integer arithmetic (integer division, modulo, and shifting have different semantics in Java and Eiffel), strings. The rest of this section describes some of these services in more detail.

JAVA_INTERFACE_PARENT is ancestor to every translated interface.

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

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

Anonymous classes are given an automatically generated name.

Arguments and attributes can be assigned to by default in Java, unlike in Eiffel where arguments are read-only and modifying attributes requires setter methods. To handle these differences, the translation $\mathcal{T}$ introduces a helper generic class *JAVA_VARIABLE* [*G*]. Instances of this class replace Java variables; assignments to arguments and attributes in Java are translated to suitable calls to the routines in the helper class.

Constructor chaining is made explicit with calls to **super**.

Field hiding is rendered by the naming scheme introduced by $\mathcal{T}$ (Section 3.4).

Field initializations and initializers are added explicitly to every constructor.

Inner classes are extracted into stand-alone classes, which can access the same outer members (features) as the original inner classes.

JavaDoc comments are ignored.

Static members. Eiffel's **once** routines can be invoked only if they belong to effective (not deferred) classes; this falls short of Java's semantics for static members of abstract classes. For each Java class C, the translation $\mathcal{T}$ introduces a class *C_STATIC* which contains all of C's static members and is inherited by the translation of C; multiple inheritance accommodates such helper classes. *C_STATIC* is always declared as effective (not deferred), so that static members are always accessible in the translation as **once** routines.

Varargs arguments are replaced by arguments of type array.

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

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

Loops. The translation of *loops* is tricky because Java allows control-flow breaking instructions such as **break**. Correspondingly, the translation of **while** loops relies on an auxiliary function ∇_W : JavaInstruction$\times\{\top,\bot\} \to$ EiffelInstruction which replicates the semantics in presence of **break** (with t $\in \{\top,\bot\}$):

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

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

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

– Exception handlers are associated to whole routines in Eiffel (**rescue** block) but to arbitrary (possibly nested) blocks in Java (**try**..**catch** blocks).

– The usage of control-flow breaking instructions (e.g., **break**) in Java's **try**.. **finally** blocks complicates the propagation mechanism of exceptions [15].

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

$$
\begin{aligned}
\nabla_M(\textbf{try } \{\text{doTry}\} \textbf{ catch } (t\ e)\ \{\text{doCatch}\}) = {} & \mathit{skipFlag} := \textbf{False} \\
& (\textbf{agent } (\mathit{args}) \textbf{ do} \\
& \quad \textbf{if not } \mathit{skipFlag} \textbf{ then} \nabla_M(\text{doTry}) \textbf{ end} \\
& \textbf{rescue} \\
& \quad \textbf{if } e.\mathit{conforms\_to}\ (\nabla_T(t)) \textbf{ then} \\
& \quad\ \nabla_M(\text{doCatch})\,;\, \mathit{Retry} := \textbf{True}\,;\, \mathit{skipFlag} := \textbf{True} \\
& \quad \textbf{else } \mathit{Retry} := \textbf{False end} \\
& \textbf{end}).\mathit{call} \\
\nabla_M(\textbf{throw } (\text{exp})) = {} & (\textbf{create } \{\mathit{EXCEPTION}\}).\mathit{raise}\ (\nabla_E(\mathit{exp}))
\end{aligned}
$$

The agent's body contains the translation of Java's try block. If executing it raises an exception, the invocation of *raise* on a fresh exception object transfers control to the **rescue** block. The **rescue**'s body executes the translation of the **catch** block only if the type of the exception raised matches that declared in the **catch** (∇_T translates Java types to appropriate Eiffel types, see Section 3.2). Executing the **catch** block may raise another exception; then, another invocation of *raise* would transfer control to the appropriate outer **rescue** block: the propagation of exceptions works similarly in Eiffel and Java. On the contrary, the semantics of Eiffel and Java diverge when the **rescue**/**catch** block terminates without exceptions. Java's semantics prescribes that the computation continues normally, while, in Eiffel, the computation propagates the exception (if *Retry* is **False**) or transfers control back to the beginning of the **agent**'s body (if *Retry* is **True**). The translation ∇_M sets *Retry* to **False** if **catch**'s exception type is incompatible with the exception raised, thus propagating the exception. Otherwise, the **rescue** block sets *Retry* and the fresh boolean flag *skipFlag* to **True**: control is transferred back to the **agent**'s body, which is however just skipped because *skipFlag* = **True**, so that the computation continues normally after the **agent** without propagating any exception.

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

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

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

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

$$
\begin{aligned}
\nabla_M(\textbf{try } \{\text{doTry}\} \textbf{ finally } \{\text{doFinally}\}) = {} & \mathit{skipFlag} := \textbf{False} \\
& (\textbf{agent } (\mathit{args})\ \textbf{do} \\
& \quad \textbf{if not } \mathit{skipFlag} \textbf{ then} \nabla_M(\text{doTry ; doFinally}) \textbf{ end} \\
& \textbf{rescue } \nabla_M(\text{doFinally}) \\
& \quad \textbf{if } \mathit{breakFlag} \textbf{ then} \\
& \quad\ \mathit{Retry} := \textbf{True}\ ;\ \mathit{skipFlag} := \textbf{True} \\
& \quad \textbf{end} \\
& \textbf{end}).\mathit{call}
\end{aligned}
$$

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

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

Assertions. Java's **assert** *exp* raises an exception if *exp* evaluates to **false**, whereas a failed **check** *exp* **end** in Eiffel sends a signal to the runtime which terminates execution or invokes the debugger. Java's assertions are therefore translated as **if not** *exp* **then** ∇_M(**throw** (**new** *AssertionError* ())) **end**.

Block locals are moved to the beginning of the current method; the naming scheme (Section 3.4) prevents name clashes.

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

∇_B(type *method* (args) { body }) = *method* (args ; predecessor: BOOLEAN): ∇_T(type) **do**
  if *predecessor* **then Precursor** (*args*, **False**)
  else ∇_M(body) **end**
end
∇_E(*method*(exp)) = *method* (∇_E(exp), **False**)
∇_E(**super**.*method*(exp)) = *method* (∇_E(exp), **True**)

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

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

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

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

```
skipFlag := False ; rethrowFlag := False
(agent (args) do
   if not skipFlag then ∇M(doTry)
   else if e.conforms_to (∇T(t)) then ∇M(doCatch) else rethrowFlag := True end end
   skipFlag := True ; ∇M(doFinally)
   if rethrowFlag and not breakFlag then (create {EXCEPTION}).raise end
 rescue if not skipFlag then skipFlag := True ; Retry := True end
 end).call
```

This translation applies uniformly to all exception-handling code and avoids duplication of the **finally** block, hence the **agent**'s body structure is more similar to the Java source. The formalization ∇_M above, however, allows for a more focused presentation and lends itself to easier formal reasoning (see Section 4.1). A correctness proof of the implementation could then establish that ∇_M and the implementation J2Eif describe translations with the same semantics.

3.2 Types and Structures

The naming scheme (Section 3.4) handles references to classes and interfaces as types; primitive types and some other type constructors are discussed here.

Primitive types with the same machine size are available in both Java and Eiffel: Java's **boolean**, **char**, **byte**, **short**, **int**, **long**, **float**, and **double** exactly correspond to Eiffel's *BOOLEAN*, *CHARACTER_32*, *INTEGER_8*, *INTEGER_16*, *INTEGER_32*, *INTEGER_64*, *REAL_32*, and *REAL_64*.

Arrays in Java become instances of Eiffel's helper *JAVA_ARRAY* class, which inherits from the standard EiffelBase *ARRAY* class and adds all missing Java functionalities to it.

Enumerations and annotations are syntactic sugar for classes and interfaces respectively extending *java.lang.Enum* and *java.lang.annotation.Annotation*.

3.3 Runtime and Native Interface

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

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

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

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

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

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

Native interface. Java Native Interface (JNI) supports calls to and from precompiled libraries from Java applications. JNI is completely independent of the rest of the Java runtime: a C **struct** includes, as function pointers, all references to native methods available through the JNI. Since Eiffel includes an extensive support to call external C code through the CECIL library, replicating JNI's functionalities in J2Eif is straightforward. The helper class *JAVA_PARENT*—accessible in every translated class—offers access to a **struct** *JNIEnv*, which contains function pointers to suitable functions wrapping the native code with CECIL constructs. This way, the Eiffel compiler is able to link the native implementations to the rest of the generated binary.

This mechanism works for all native JRE libraries except for the Java Virtual Machine (*jvm.dll* or *jvm.so*), which is specific to the implementation (OpenJDK in our case) and had to be partially re-implemented for usage within J2Eif. The current version includes new implementations of most JVM-specific services, such as *JVM_FindPrimitiveClass* to support reflection or *JVM_ArrayCopy* to duplicate array data structures, and verbatim replicates the original implementation of all native methods which are not JVM-specific (such as *JVM_CurrentTimeMillis* which reads the system clock). The experiments in Section 4 demonstrate that the current JVM support in J2Eif is extensive and sufficient to translate correctly many Java applications.

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

3.4 Naming

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

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

- η normalizes a name by successively (1) replacing all "_" with "_1", (2) replacing all "." with "_", and (3) changing all characters to uppercase—for example, η(*java.lang.String*) is *JAVA_LANG_STRING*;
- $\phi(n)$ denotes the fully-qualified name of the item n—for example, ϕ(*String*) is, in most contexts, *java.lang.String*;
- $\lambda(v)$is an integer denoting the nesting depth of the block where$v$ is declared—for example, in the method **void** *foo*(**int** *a*){**int** *b*; **for**(**int** *c*=0;...)...}, it is $\lambda(a) = 0$, $\lambda(b) = 1$, $\lambda(c) = 2$.

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

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

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

4 Evaluation

This section briefly discusses the correctness of the translation $\mathcal{T}$ (Section 4.1); evaluates the usability of its implementation J2Eif with four case studies (Section 4.2); and concludes with a discussion of open issues (Section 4.3).

4.1 Correctness of the Translation

While the formalization of $\mathcal{T}$ in the previous sections is not complete and overlooks some details, it is useful to present the translation clearly, and it even helped the authors find a few errors in the implementation when its results did not match the formal model. Assuming an operational semantics for Java and Eiffel (see [17]), one can also reason about the components of $\mathcal{T}$ formalized in Section 3 and increase the confidence in the correctness of the translation. This section gives an idea of how to do it; a more accurate analysis would leverage a proof assistant to ensure that all details are taken care of appropriately.

The operational semantics defines the effect of every instruction I on the program state: $\sigma \xrightarrow{I} \sigma'$ denotes that executing I on a state σ transforms the state to σ'. The states σ, σ' may also include information about exceptions and non-terminating computations. While a Java and an Eiffel state are in general different, because they refer to distinct execution models, it is possible to define an equivalence relation $\simeq$ that holds for states sharing the same “abstract”

values [17], which can be directly compared. With these conventions, it is possible to prove correctness of the formalized translation: the effect of executing a translated Eiffel instruction on the Eiffel state replicates the effect of executing the original Java instruction on the corresponding Java state. Formally, the correctness of the translation of a Java instruction I is stated as: "For every Java state σ_J and Eiffel state σ_E such that $\sigma_J \simeq \sigma_E$, if $\sigma_J \xrightarrow{I} \sigma'_J$ and $\sigma_E \xrightarrow{\nabla_M(I)} \sigma'_E$ then $\sigma'_J \simeq \sigma'_E$."

The proof for the the Java block B: **try** {doTry} **catch** (t e) {doCatch}, translated to $\nabla_M(B)$ as shown on page 7, is now sketched. A state σ is split into two components $\sigma = \langle v, e \rangle$, where e is ! when an exception is pending and $\star$ otherwise. The proof works by structural induction on B; all numeric references are to Nordio's operational semantics [17, Chap. 3]; for brevity, consider only one inductive case.

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

4.2 Experiments

Table 2 shows the results of four experiments run with J2Eif on a Windows Vista machine with a 2.66 GHz Intel dual-core CPU and 4 GB of memory. Each experiment consists in the translation of a system (stand-alone application or library). Table 2 reports: (1) the size in lines of code of the source (J for Java) and transformed system (E for Eiffel); (2) the size in number of classes; (3) the source-to-source compilation time (in seconds) spent to generate the translation ($\mathcal{T}$, which does not include the compilation from Eiffel source to binary); (4) the size (in MBytes) of the standard (s) and optimized (o) binaries generated by EiffelStudio; (5) the number of dependent classes needed for the compilation (the SWT snippet entry also reports the number of SWT classes in parentheses). The rest of the section discusses the experiments in more detail.

	Size (locs)		#Classes		Compilation (sec.)	Binary Size (MB)		#Required Classes
	J	E	J	E	$\mathcal{T}$	s	o	
HelloWorld	5	92	1	2	1	254	65	1208
SWT snippet	34	313	1	6	47	318	88	1208 (317)
java.util.∗	51,745	91,162	49	426	7	254	65	1175
java.io tests	11,509	28,052	123	302	6	255	65	1225

Table 2. Experimental results.

HelloWorld. The *HelloWorld* example is useful to estimate the minimal number of dependencies included in a stand-alone application; the size of 254 MB (65 MB optimized) is the smallest footprint of any application generated with J2Eif.

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

java.util.∗ classes. Table 3 reports the results of performance experiments on some of the translated version of the 49 data structure classes in *java.util*. For each Java class with an equivalent data structure in EiffelBase, we performed tests which add 100 elements to the data structure and then perform 10000 removals of an element which is immediately re-inserted. Table 3 compares the time (in ms) to run the test using the translated Java classes (column 2) to the performance with the native EiffelBase classes (column 4).

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

Table 3. Performance of translated *java.util* classes.

The overhead introduced by some features of the translation adds up in the tests and generates the significant overall slow-down shown in Table 3. The features that most slowed down the translated code are: (1) the indirect access to fields via the *JAVA_VARIABLE* class; (2) the more structured (and slower) translation of control-flow breaking instructions; (3) the handling of exceptions with agents (whose usage is as expensive as method call). Applications that do not heavily exercise data structures (such as GUI applications) are not significantly affected and do not incur a nearly as high overhead.

java.io test suite. The part of the Mauve test suite [11] focusing on testing input/output services consists of 102 classes defining 812 tests. The tests with J2Eif excluded 10 of these classes (and the corresponding 33 tests) because they relied on unsupported features (see Section 4.3). The functional behavior of the tests is identical in Java and in the Eiffel translation: both runs fail 25 tests and pass 754. Table 4 compares the performance of the test suite with Java against its Eiffel translation; the two-fold slowdown achieved with optimizations is, in all, usable and reasonable—at least in a first implementation of J2Eif.

	Overall time (s)	Average time per test (ms)	Slowdown
Java	4	5	1
Eiffel standard	21	27	5.4
Eiffel optimized	9	11	2.2

Table 4. Performance in the *java.io* test suite.

4.3 Limitations

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

Unicode strings. J2Eif only supports the ASCII character set; Unicode support in Eiffel is quite recent.

Serialization mechanisms are not mapped adequately to Eiffel's.

Dynamic loading mechanisms are not rendered in Eiffel; this restricts the applicability of J2Eif for applications heavily depending on this mechanism, such as J2Eif itself which builds on the Eclipse framework.

Soft, weak, and phantom references are not supported, because similar notions are currently not available in the Eiffel language.

Readability. While the naming scheme tries to strike a good balance between readability and correctness, the generated code may still be less pleasant to read than in a standard Eiffel implementation.

Size of compiled code. The generated binaries are generally large. A finer-grained analysis of the dependencies may reduce the JRE components that need to be included in the compilation.

5 Related Work

There are two main approaches to reuse implementations written in a "foreign" language within another "host" language: using wrappers for the components written in the "foreign" language and bridging them to the rest of the application written in the "host" language; and translating the "foreign" source code into functionally equivalent "host" code.

Wrapping foreign code. A wrapper enables the reuse a foreign implementation through the API provided by a bridge library [5, 4, 19, 13]. This approach does not change the foreign code, hence there is no risk of corrupting it or of introducing inconsistencies; on the other hand, it is usually restrictive in terms of the type of data that can be retrieved through the bridging API (for example, primitive types only). J2Eif uses the wrapping approach for Java's native libraries (Section 3.3): the original Java wrappers are replaced by customized Eiffel wrappers.

Translating foreign code. Industrial practices have long encompassed the manual, systematic translation of legacy code to new languages. More recently,

researchers proposed semi-automated translation for widely-used legacy programming languages such as COBOL [2, 14], Fortran-77 [1, 21], and C [23]. Other progress in this line has come from integrating domain-specific knowledge [6], and testing and visualization techniques [18] to help develop the translations.

Other related efforts target the transformation of code into an extension (superset) of the original language. Typical examples are the adaptation of legacy code to object-oriented extensions, such as from COBOL to OO-COBOL [16, 20, 22], from Ada to Ada95 [10], and from C to C++ [9, 24]. Some of such efforts try to go beyond the mere execution of the original code by refactoring it to be more conforming to the object-oriented paradigm; however, such refactorings are usually limited to restructuring modules into classes.

As far as fully automated translations are concerned, compilation from a high-level language to a low-level language (such as assembly or byte-code) is of course a widespread technology. The translation of a high-level language into another high-level language with different features—such as the one performed by J2Eif—is much less common; the closest results have been in the rewriting of domain-specific languages, such as TXL [3], into general-purpose languages.

Google web toolkit [7] (GWT) includes a project involving translation of Java into JavaScript code. The translation supports running Java on top of JavaScript, but its primary aims do not include readability and modifiability of the code generated, unlike the present paper's translation. Another relevant difference is that GWT's translation lacks any formalization and even the informal documentation does not detail which features are not perfectly replicated by the translation. The documentation warns the users that "subtle differences" may exist,[4] but only recommends testing as a way to discover them.

6 Conclusions

This paper presented a translation $\mathcal{T}$ of Java programs into Eiffel, and its implementation in the freely available tool J2Eif [8]. The formalization of $\mathcal{T}$ built confidence in its correctness; a set of four experiments of varying complexity tested the usability of the implementation J2Eif.

Future work includes more tests with applications from different domains; the extension of the translation to include the few aspects currently unsupported (in particular, Unicode strings and serialization); and the development of optimizations for the translation, to make the code generated closer to original Eiffel implementations.

Acknowledgements. Thanks to Mike Hicks and Bertrand Meyer for their support and advice, and to Louis Rose for comments on a draft of this paper.

References

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

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

2. G. Canfora, A. Cimitile, A. de Lucia, and G. A. D. Lucca. A case study of applying an eclectic approach to identify objects in code. In *IWPC*, pages 136–143, 1999.
3. J. R. Cordy. Source transformation, analysis and generation in TXL. In *PEPM*, pages 1–11, 2006.
4. 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*, pages 122–129, 1997.
5. W. C. Dietrich, Jr., L. R. Nackman, and F. Gracer. Saving legacy with objects. *SIGPLAN Not.*, 24(10):77–83, 1989.
6. H. Gall and R. Klosch. Finding objects in procedural programs: an alternative approach. In *WCRE*, pages 208–216, 1995.
7. Google Web toolkit. `http://code.google.com/webtoolkit/`, 2010.
8. J2Eif. The Java to Eiffel translator. `http://jaftec.origo.ethz.ch`, 2010.
9. K. Kontogiannis and P. Patil. Evidence driven object identification in procedural code. In *STEP*, pages 12–21, 1999.
10. A. Llamosí and A. Strohmeier, editors. *Reliable Software Technologies–Ada-Europe*, volume 3063 of *LNCS*. Springer, 2004.
11. Mauve project. `http://sources.redhat.com/mauve/`, 2010.
12. D. Mcilroy. Mass-produced software components. In *ICSE*, pages 88–98, 1968.
13. B. Meyer. The component combinator for enterprise applications. *JOOP*, 10(8):5–9, 1998.
14. R. Millham. An investigation: reengineering sequential procedure-driven software into object-oriented event-driven software through UML diagrams. In *COMPSAC*, pages 731–733, 2002.
15. P. Müller and M. Nordio. Proof-Transforming Compilation of Programs with Abrupt Termination. In *SAVCBS '07*, pages 39–46, 2007.
16. P. Newcomb and G. Kotik. Reengineering procedural into object-oriented systems. In *WCRE*, pages 237–249, 1995.
17. M. Nordio. *Proofs and Proof Transformations for Object-Oriented Programs*. PhD thesis, ETH Zurich, 2009.
18. M. Postema and H. W. Schmidt. Reverse engineering and abstraction of legacy systems. *Informatica*, pages 37–55, 1998.
19. M. A. Serrano, D. L. Carver, and C. M. de Oca. Reengineering legacy systems for distributed environments. *J. Syst. Softw.*, 64(1):37–55, 2002.
20. H. Sneed. Migration of procedurally oriented cobol programs in an object-oriented architecture. In *Software Maintenance*, pages 105–116, 1992.
21. G. V. Subramaniam and E. J. Byrne. Deriving an object model from legacy Fortran code. *ICSM*, pages 3–12, 1996.
22. T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In *WCRE*, pages 24–32, 1997.
23. 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.
24. Y. Zou and K. Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *APSEC*, pages 390–399, 2001.