2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

Understanding and Fixing Multiple Language Interoperability Issues: The C/Fortran Case

Nawrin Sultana, Justin Middleton, Jeffrey Overbey, Munawar Hafiz
Auburn University
Department of Computer Science and Software Engineering
Auburn, AL, USA
{nzs0034,jam0058,joverbey}@auburn.edu, munawar.hafiz@gmail.com

ABSTRACT

We performed an empirical study to understand interoperability issues in C and Fortran programs. C/Fortran interoperability is very common and is representative of general language interoperability issues, such as how interfaces between languages are defined and how data types are shared. Fortran presents an additional challenge, since several ad hoc approaches to C/Fortran interoperability were in use long before a standard mechanism was defined. We explored 20 applications, automatically analyzing over 12 million lines of code. We found that only 3% of interoperability instances follow the ISO standard to describe interfaces; the rest follow a combination of compiler-dependent ad hoc approaches. Several parameters in cross-language functions did not have standards-compliant interoperable types, and about one-fourth of the parameters that were passed by reference could be passed by value. We propose that automated refactoring tools may provide a viable way to migrate programs to use the new interoperability features. We present two refactorings to transform code for this purpose and one refactoring to evolve code thereafter; all of these are instances of multiple language refactorings.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.3.m [**Programming Languages**]: Miscellaneous

Keywords

C, Fortran, language interoperability, polyglot, refactoring

1. INTRODUCTION

Modern programs are often polyglot—written in multiple languages. The need for multiple languages arises because each language may be specifically designed for a problem domain. A recent study reported that about 96% of programs in GitHub were written using at least two languages [37].

ICSE '16, May 14-22, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2884781.2884858

Tomassetti and colleagues [38] identified six multiple language interoperability patterns. Five are about variants of interaction in which one of the languages represents data, e.g., an identifier in an XML configuration file containing the qualified name of a Java class (*Shared ID* [38]), an XML file containing configuration data that is used by a Java program (*Data Loading* [38]), etc. The survey of GitHub repositories identified these five kinds of interactions to be the most common [37]. However, the sixth kind of interaction, in which a program written in one language executes another program written in a different language (*Execution* [38]), raises more issues about comprehending, analyzing, and evolving programs. This paper concentrates on this specific interoperability pattern of polyglot programs.

Although a common phenomenon, issues of multiple language interoperability have not been explored in detail. Tan and Croft [35] studied Java and C interoperability using JNI (Java Native Interface) and identified categories of security problems. The interoperability issues were studied manually with the support of simple lexical search; 93% of the results were false positives [16]. Most problems identified in the study originate from misunderstanding the semantics of interoperability. Several papers have explored the program comprehension burden of polyglot programs [9,18,22,25] and proposed ways to improve the situation by linking artifacts across the language boundaries.

This paper describes a case study on interoperability issues considering C and Fortran programs. Interoperability between C and Fortran is featured in many legacy programs because of the popularity of those programming languages in specific problem domains. It manifests the typical issues arising in general multiple language interoperability scenarios—issues such as interface discovery, datatype sharing and memory management, data sharing semantics, etc. Additionally, the Fortran language has evolved through several versions (Fortran 90, 95, 2003, and 2008 are the most recent); thus, interoperable programs have had the opportunity to evolve as well. The ad hoc style of interoperability used in older Fortran programs depends on the compiler. C interoperability was standardized in Fortran 2003 [14], finally making it possible to write C-interoperable Fortran code that was also portable between platforms and between compilers. This was a reactive change to the ISO standard, made specifically because the Fortran community had demonstrated a need for it. Unfortunately, this means the language was "fixed" long after developers encountered the problems involved. This motivates the need for a tool that can reliably upgrade legacy code to make it standards-

compliant, since large-scale rewrites are often not feasible. We studied 20 polyglot programs—automatically analyzing over 12 million lines of code—to understand the issues of C and Fortran interoperability. We found that very few programs use the portable C interoperability mechanism introduced in Fortran 2003 (4 out of 20 programs, 27 out of 823 cross-language function calls, ≈ 3%). We also found that some datatypes supported by the ad hoc interoperability mechanisms are not supported by the current Fortran 2003 interoperability standard. For example, unsigned datatypes are shared between programs in an unsafe manner that may lead to subtle bugs or even security problems [21]. Finally, we found that parameters of many cross-language function calls are passed by reference even though they are only used in a read-only manner (154 parameters out of 683, 154/683 ≈ 23%), thus creating the potential for security problems by violating the principle of least privilege [28].

The findings from the case study guide the design of a refactoring-based approach to fix the interoperability issues. We introduce three refactorings. Two of them modify programs that follow old C and Fortran interoperability conventions to take advantage of the new interoperability standards introduced in the ISO Fortran 2003 standard [14]. A Convert To Standard Data Type (CSDT) refactoring identifies variables that are shared between C and Fortran and converts the Fortran declarations to use interoperable types compliant with the ISO standard. A Transform To Standard Interface (TSI) refactoring identifies a C function called from Fortran and introduces an interface block into the Fortran code, which provides a prototype for the function and allows the Fortran compiler to type-check the arguments at its call sites. The third refactoring is a variation of the Change Function Signature (CFS) refactoring that applies to multiple languages. It helps evolve the interface between C and Fortran programs after they have been updated to follow the ISO Fortran 2003 standard.

These three are examples of multiple language program transformations. Researchers have explored similar program transformations primarily for renaming identifiers [7,22,32]. In contrast, our refactorings are more complex: they require a tool capable of parsing and analyzing types in both the host and foreign languages, they depend on data flow and dependence analysis to determine a safe transformation option, and they ultimately help programmers convert programs using ad hoc, compiler-dependent, and potentially unsafe interoperability conventions into programs that rely only on features supported by the ISO standard.

This paper makes the following contributions:

- To the best of our knowledge, this is the first paper to present *empirical* data studying the interoperability issues of multiple language programs (§§ 4–5).
- The paper explores how the evolution of interoperability features affects legacy code and introduces a refactoring-based solution to evolve programs (§§ 5–6).
- The paper introduces multi-language program transformations based on syntax- and semantics-aware analyses of both the host and foreign languages (§§ 6–7).

2. MOTIVATING EXAMPLE

One set of programs in our test corpus is a repository of networking and data structure code from the Numerical Algorithms Group (NAG). It contains 18 C and Fortran interoperable functions in its 11 C files and 207 Fortran files (total 86 KLOC). The following interoperability example is based on the *tcp_connect_* function defined in `clientc.c` and called from `client.F90`. The *client_connect* subroutine in the Fortran program calls the C function:

```
subroutine client_connect(handle, host_name,
    host_port, error)
  integer, intent(out) :: handle
  character*(*), intent(in) :: host_name
  integer, intent(in) :: host_port
  logical, intent(out) :: error
  ...
  call tcp_connect(fd_table(handle)%fd%socket,
      host_name, host_port, error)
  ...
  return
end subroutine client_connect
```

The corresponding C function definition is in client.c.

```
#define tcp_connect tcp_connect_
...
void tcp_connect (sock, host_name, port, error,
    host_name_len)
  int *sock, *port, *error;
  char *host_name;
  int host_name_len;
  {
    ...
  }
```

There are several issues with this code. First, the definition in C has a trailing underscore defined through the macro. Adding a trailing underscore is an ad hoc mechanism of specifying an interoperable C function. Shenkin [31] calls this convention `BIND_`; we will adopt that terminology in this paper. The Fortran compiler adds an underscore for the foreign function to be called; the C definition matches that. But this may not be portable for another compiler. Second, the function call in the Fortran code has four parameters, but the C function definition has five parameters. The Fortran compiler adds the additional parameter (`host_name_len`)—in general, it adds one parameter for each string parameter that is passed. Again, this is compiler-dependent. If the passed parameter is a string literal, the length parameter may not be passed. The Fortran 2003 standard removes the need for the extra parameter. Third, the `port` parameter is passed as a reference. Although it is not shown here, the *tcp_connect_* function uses it in a read-only manner. Passing it as a reference violates the least privilege principle [28], which may lead to subtle bugs and even security problems [21].

The case study (§§ 4–5) presents empirical data on C/Fortran interoperability issues. Then, we describe three refactorings to fix the interoperability issues (§ 6). Convert To Standard Data Type updates the shared data types to follow the standard. Transform To Standard Interface updates the interface description and fixes the parameter passing styles. Once updated, Change Function Signature can be used to evolve the interoperable code.

3. BACKGROUND

This section describes previous work on understanding language interoperability issues, creating interoperability standards, comprehending and analyzing multi-language programs, and transforming programs written in multiple programming languages.

3.1 Understanding Multiple Language Issues

Researchers have identified the importance of multi-language programs in software evolution research [5,24], yet there has not been any empirical study to understand the issues in interoperability semantics. Vetro and colleagues [39] studied commits in the Apache Hadoop repository. They distinguished commits that touched translation units of multiple programming languages (cross-language commit). They found that most common cross-language commits in Hadoop are between C/Java and a language for data representation (XML). However, they reported that a commit affecting C and Java is more likely to be error prone, perhaps because of the underlying complexity. Tomassetti and colleagues [38] reported that the most common multi-language interoperability patterns are between a high level programing language and a language for data representation; these were featured in five out of their six interoperability patterns. They also found that the interaction between two high level languages is more complex.

Tan and Croft [35] studied a portion of the native code in Sun's JDK 1.6 to understand the security issues in Java and C interoperability. They distinguished seven categories of security problems. Most of these problems originate because developers misunderstand interoperability issues that may lead to buffer overflows. However, some of these security problems may arise from the issues of multi-language interoperability. For example, C pointers are often casted to Java integers so that they can be passed through Java code, but this may cause security problems if an attacker can inject arbitrary integer values that will be interpreted as pointers [12]. Our study reports similar problems, but its scope is broader—we wanted to explore how the interfaces are described and how the data types are shared in general.

3.2 Interoperability Standards

Many programming languages provide a foreign function interface(FFI) [3, 6] that allows programs to call routines or use services written in another language. Java has the Java Native Interface (JNI) [17], which allows Java code to interoperate with code written in languages like C and C++.

This paper focuses on interoperability between C and Fortran. Before the Fortran 2003 standard [1], C and Fortran programs interacted in ad hoc ways that depended on the specific C and Fortran compilers. The names of interoperable C functions needed to be changed according to the Fortran compiler's naming convention; a popular convention ("`BIND_`") was for the compiler to append a trailing underscore to a Fortran function's name [31]. Parameters were usually passed by reference, but some compilers provided a non-standard function `%VAL()` that allowed parameters to be passed by value. Passing character strings added additional complexity. Most Fortran compilers passed the string length by value using a hidden argument, which had to be explicitly added to the corresponding C function [31].

Fortran 2003 standardized interoperability with C. It provides a `BIND(C)` attribute to specify C linkage and a module called `ISO_C_BINDING`. Fortran 90 introduced `KIND` parameters to select different machine representations for each of the intrinsic data types (e.g., 4- vs. 8-byte integers); Fortran 2003's `ISO_C_BINDING` module includes named constants used to specify `KIND` parameters for C-interoperable types. The 2003 standard also eliminated the need for C function names to have a trailing underscores appended, and it introduced an attribute `VALUE` for passing parameters by value. Finally, it also resolved the issues with character strings, eliminating the need for hidden length arguments.

Chivers and Sleightholme [8] reported in 2007 that most compilers supported the Fortran 2003 standard, but code written following the ad hoc C/Fortran interoperability mechanisms caused unexpected runtime errors when compiled by Fortran 2003 compilers. In this paper, we introduced two automated refactorings that transform programs written following the old ad hoc C and Fortran interoperability mechanism to Fortran 2003 standard interoperability.

3.3 Comprehending Polyglot Programs

Several researchers have aimed to reduce the program comprehension burden for developers of multi-language programs. Their general approach is to use a model for representing facts about different programming languages and link facts between the host and foreign languages. Linos [18] created an early prototype to show the dependencies among multiple programming paradigms. Linos and colleagues [19] later explored a more sophisticated prototype to demonstrate host-to-foreign language dependencies between Java and C++ source code. A later work explored an intermediate representation to determine the dependencies more efficiently [20].

Mayer and Schroeder [22] not only captured the dependencies between languages ("semantic links") in an explicit structure (XLL, acronym for Cross Language Links), but also demonstrated how such links can be used to perform simple refactorings. A somewhat different approach to facilitate multiple language comprehension is DSketch. Cossette and colleagues [9] used a pattern matching approach to link artifacts among languages. However, developers have to write these pattern specifications as an overlay on the underlying code.

The models created to represent different programming languages are mostly different, since different languages follow different programming paradigms (object-oriented vs. functional vs. database-specific, etc.,). To create the models, researchers had to separately analyze the programs written in different languages, then link them at a later stage. In contrast, our empirical study analyzes C and Fortran programs together to extract facts; a similar approach is required to support complex multi-language refactorings.

3.4 Multiple Language Refactorings

Strein and colleagues [32] first explored refactoring multi-language programs. Interfaces were described using a common meta-model; the transformations were performed by separate language adapters. They later explored how to efficiently store and evolve the common meta-model to support the transformations [33]. They implemented a Rename refactoring on a combination of Visual Basic, C#, and J#. The renaming mechanism was based on searching identifiers and matching their semantics with the identifier to be renamed. The work on creating a common meta-model for multi-language refactorings followed from prior work by Tichelaar and colleagues [36].

Chen and Johnson [7] extended Eclipse's Java Rename refactoring to support the refactoring of XML identifiers used by the Struts, Hibernate, and Spring frameworks. Their work was not based on a common meta-model of the two languages. They pointed out that finding a general solu-

tion based on a common meta-model for multiple language refactorings is a hard problem; instead, researchers should focus on how to support the common refactorings individually. Kempf and colleagues [15] followed a similar approach to support the RENAME refactoring in Eclipse for Java and Groovy. Both of these works are based on the refactoring participant mechanism [10] introduced in Eclipse 3.2, which follows the *Observer* pattern [11]. The subject role is played by the refactoring operation on Java files, while the observers are refactoring operations to be performed on the XML files.

Schink and colleagues [30] implemented multiple language versions of the RENAME METHOD and PUSH DOWN METHOD refactorings for Java, Hibernate, and SQL. They also introduced a specific refactoring for relational databases [29]. They concluded that following a common meta-model is not feasible across all combinations. Mayer and Schroeder [22] used their analysis mechanism for polyglot programs (§ 3.3) to support renaming across Java, Ruby, and XML.

The multi-language refactorings in this paper are more complex than a RENAME-like refactoring. They require type analysis as well as data flow and dependence analysis in the host and foreign languages.

4. INTEROPERABILITY STUDY

4.1 Research Questions

In our empirical study of C/Fortran interoperability, we explore how the interface between the languages is defined, which data types are shared, and how the data types are shared. Since the interface between Fortran and C was standardized in Fortran 2003 [14], we explore whether developers have been following this approach.

We ask three research questions:

RQ1: Interface. How are interfaces declared in the test programs? How often are ad hoc interoperability mechanisms used?

RQ2: Shared Data Types. What data types are shared between C and Fortran code in our corpus? How are data types supported by ad hoc interoperability mechanisms handled differently by the Fortran 2003 standard?

RQ3: Parameter Passing Mechanism. How often are arguments passed by reference vs. by value? How often are arguments passed by reference when they could be passed by value instead?

4.2 Test Corpus

We explored 20 applications that contain C and Fortran functions calling each other. Five applications were collected from the test suite of Photran, which provides Fortran support in Eclipse. We selected applications that contained C and Fortran code. Nine applications were collected from a paper on global climate models [23]. Finally, six applications were collected by searching GitHub for Fortran applications, then selecting the ones that also contain C. Table 1 lists the applications along with the number of C and Fortran files and the size of the applications. We automatically analyzed over 12 million lines of code from over 30,000 files.

Of the 20 applications, 9 modeled global climate systems for use in research and forecasts; CCSM4, CESM, CFS, CMCC, G, and MODEL are from the Coupled Model Intercomparison Project (CMIP5) [40]. Three other projects were scientific simulations: Elegant (ELE), NWChem 6.5 (NWC), and MagneticModel (MAG). Elegant began as a particle generator and tracker and has since grown to include much more; we found a version which included tools written as polyglot programs in the mid-1990s. NWChem is a computational chemistry package; we used the 2014 release. MagneticModel was a repository of utilities for the World Magnetic Model [26] in 2010, which are simulation programs for the Earth's magnetic field. This project had been steadily updated between 2010 and 2015.

Table 1: Applications Used In The Study

Name of Applications (ID Used to Refer to it)	# of C Files	# of Fortran Files	KLOC
Asteriods (AST)	1	2	5.6
ccsm3 (CCSM3)	62	791	405.0
ccsm4 (CCSM4)	37	1,380	865.0
cesm (CESM)	28	1,976	1,418.0
cfs_v2 (CFS)	95	1,144	948.0
CM2.5_and_FLOR (CM)	24	627	846.0
CMCC-CESM (CMCC)	15	1,026	410.0
CompiledExamples (COM)	2	2	0.2
Elegant (ELE)	156	6	110.0
GEOSagcm-Eros_7_24 (G)	14	589	375.0
Loop-with-bitmask (LP)	5	1	0.5
MagneticModel (MAG)	8	4	13.0
Mechanic (MECH)	56	1	14.0
ModelE2 (MODEL)	2	551	438.0
Nwchem-6.5 (NWC)	1,545	17,054	6,013.0
rimbaud (RIM)	18	4	3.6
solsys (SOL)	30	40	15.0
WRFV3 (WRF)	149	700	871.0
www.NAG (WWW)	11	207	86.0
Yoda (YOD)	7	2	7.5
Total	2,546	28,043	12,844.0

The other collected applications are from GitHub and other repositories on the web, and as such, their functions are not closely related. The interoperable functions in COM and RIM, both authored in 2014, were intended explicitly as small examples exploring interoperability in C, Fortran, and in some cases R. MECH (released around 2012) also follows this pattern; being a task management system for simulations, it allows extensibility with other languages and includes examples of C/Fortran interoperability to illustrate it. AST is a 1996 implementation of the Asteroids video game using both C and Fortran to demonstrate an interface with the X Window System, whereas YOD is a deconvolution algorithm, released around 2011, for the Yorick language. LP is a 2012 accumulation operator for sparse arrays. SOL includes code to identify and remove radio interference for telescopes with some parts of the code dating to the late 1990s. WWW is miscellany from the Numerical Algorithm Group's repository. It contains networking and data structure code written throughout the 1990s.

4.3 Data Collection Mechanism

For RQ1, we had to search for four different ways of describing the interfaces: (1) appending an underscore to interoperable C function names (known as the `BIND_` convention), (2) using `external`, (3) using a macro, and (4) using the C binding interface described in Fortran 2003. We used a combination of *grep* and *ctags* tools for this.

The *ctags* tool generates an index file of names (including functions) found in source and header files. To find the interface definition following the `BIND_` convention, we collected all the function names with a trailing underscore from the

Table 2: Interoperability Interfaces

App. ID	Using 2003 Standard	Using BIND_ Convention	Using external Declaration	Using Macro	Other
AST	0	41	0	0	0
CCSM3	0	70	4	26	0
CCSM4	0	0	22	15	0
CESM	21	0	32	3	0
CFS	0	2	0	0	0
CM	0	13	0	0	0
CMCC	0	7	2	5	0
COM	0	0	0	0	2
ELE	0	3	0	0	0
G	0	7	1	0	0
LP	0	3	0	0	0
MAG	0	1	0	0	0
MECH	2	0	0	0	0
MODEL	1	1	0	0	0
NWC	0	224	65	211	0
RIM	0	3	0	0	0
SOL	0	1	0	0	0
WRF	3	12	0	0	0
WWW	0	15	3	0	0
YOD	0	2	0	0	0
Total	27	405	129	260	2

C side and then looked for matching function calls from the Fortran side. For the other three approaches, we started on the Fortran side. We used *grep* to look for the `external` keyword, patterns describing the macros, and the `iso_c_binding` and `BIND(C)` keywords. Then, we searched for the same binding label in C code for each of the matching entities. This process was run by automated scripts.

RQ2 focuses on the data types shared between C and Fortran. We used the type analysis support in OpenRefactory/C [13] to determine the types of all function parameters.

For RQ3, we explored whether the C function definitions expect the parameters to be passed by value or by reference. Again using OpenRefactory/C, we used type analysis to distinguish pointer types and performed a reaching definitions analysis on the parameters (and their aliases) to determine if the passed parameters were redefined inside the function. We used these results to determine which parameters could be converted to pass-by-value on the Fortran side. We only considered basic (i.e., intrinsic) types, like integers. We assumed complex types such as arrays and structures would always be passed by reference.

5. UNDERSTANDING C/FORTRAN INTEROPERABILITY

This section describes the results from our study and how the results contribute to design decisions for the refactorings.

5.1 RQ1: Interface

How are interfaces declared in the test programs? How often are ad hoc interoperability mechanisms used?

Key Result: Most applications containing interoperable code use ad hoc interoperability mechanisms, making the code compiler-dependent.
Design Decision: Code using the `BIND_` and `external` conventions can be refactored to follow the interoperability mechanism proposed in Fortran 2003.

Table 2 lists four approaches to describe the interfaces of interoperable functions. The most up-to-date approach is to follow the interface specification mechanism described in the ISO Fortran 2003 standard [14]. One ad hoc (compiler-dependent) approach is to append an underscore to the C routine name for interfacing with a Fortran routine (`BIND_` convention.). Another approach is to use an `external` declaration in the Fortran code, while the C routine uses the compiler-specific underscore. Finally, a programmer can use a macro in C code to define interoperable functions.

There were 823 interoperable procedures in the applications. Most of them ($\approx$ 49%) followed the `BIND_` convention for interfacing. About 16% used `external`. Only a few instances ($\approx$ 3%) followed the Fortran 2003 interoperability standard.

Most of the applications were written and/or evolved after the Fortran 2003 standard (§ 4.2). Yet, only four applications contain some instances following the Fortran 2003 standard. All of these applications contain many other interoperability instances that follow a combination of old techniques. In fact, nine applications contain interoperable C and Fortran functions that follow multiple interoperability approaches. Perhaps, the code was written by different programmers and evolved through a long period of time.

The Fortran 2003 standard requires that an elaborate interface be declared for interoperable procedures. The following listing from MECH shows an interface definition for a function named *cfunc*.

```
// Fortran interface and call:
  interface
   function cfunc(n, x) result(cval) bind(c)
    use iso_c_binding
    implicit none
    integer(c_int), intent(in), value :: n
    real(c_double), intent(inout), dimension(24) :: x
    integer(c_int) :: cval
   end function cfunc
  end interface
   ...
  pstatus = cfunc(n,x)

// C function:
  int cfunc(int n, double *x) {
    ...
  }
```

The same declaration can be made more concisely using the `BIND_` convention or the `external` keyword, but it comes at the cost of portability. For example, the `BIND_` approach for specifying the previous example may look like this:

```
// Fortran interface and call:
  integer :: cfunc, pstatus
  integer :: n
  real(kind=8), dimension(24) :: x
  ...
  pstatus = CFUNC(n, x)

// C function:
  int cfunc_(int *n, double *x) {
    ...
  }
```

Table 2 shows that many interoperable routines were written using C macros. This convention was used to call external library routines for specific libraries, e.g., Performance Application Programming Interface (PAPI), Message Passing Interface (MPI), etc. The programmers of applications in our corpus used two types of macros—`FORT_NAME` and `F77_FUNC_`.

Table 3: Shared Data Types (Only Parameters In Functions Following Ad Hoc Interoperability Standard)

App. ID	Shared Intrinsic Type		Shared Char Type	Shared Array Type	Shared Pointer			Shared Derived Type	Shared Logical Type
	int, float, double type	unsigned type			single pointer	double pointer	function pointer		
AST	52	0	0	6	21	27	0	6	1
CCSM3	126	4	5	12	0	0	0	86	6
CCSM4	27	2	8	11	0	0	0	0	0
CESM	33	4	9	0	0	2	0	0	0
CFS	5	0	4	0	0	0	0	0	0
CM	27	0	0	79	0	0	0	0	8
CMCC	0	0	4	0	0	2	0	5	0
COM	6	0	0	0	0	0	0	0	0
ELE	102	0	0	33	0	0	0	0	0
G	13	1	0	8	0	0	0	0	0
LP	6	0	3	3	0	0	0	0	0
MAG	1	0	1	11	0	0	0	0	0
MODEL	2	0	0	0	0	0	0	0	0
NWC	355	7	93	70	10	0	0	0	6
RIM	6	0	0	2	0	0	0	0	0
SOL	4	0	0	1	0	0	0	0	0
WRF	7	32	1	0	0	0	0	0	0
WWW	18	0	15	2	9	0	5	4	6
YOD	35	0	2	5	0	0	0	0	1
Total	825	50	145	243	40	31	5	101	28
Fortran 2003 Conversion Approach	Add compatible name constant and attribute	Disallow (Potentially Unsafe: Cast to signed intrinsic)	Remove extra length from C function	Add compatible name constant	Add `c_ptr` for void pointer + cast to integer	Our transformation don't support	Use `c_funptr` (Not supported yet)	Add `BIND` attribute and name constant for members	Add compatible `C_BOOL` type

Macros were typically used to call external library routines. Only in two instances in the CompiledExamples-master (COM) application were macros were used to call a user-defined routine in Fortran from C. We distinguished these instances in the "other" category.

Program interfaces following the `BIND_` convention or the `external` keyword can be transformed to Fortran 2003-style interface definitions. Interfaces defined by macros could also be converted to Fortran 2003-style interface definitions, but we did not implement this, since the macros were specific to a single application. This leads to two variants of the Transform To Standard Interface refactoring (§ 6).

5.2 RQ2: Shared Data Types

What data types are shared between C and Fortran code in our corpus? How are data types supported by ad hoc interoperability mechanisms handled differently by the Fortran 2003 standard?

Key Result: Many data type interoperability issues were resolved by the Fortran 2003 standard, although some data types are not supported by the standard.
Design Decision: Treat each of the types separately and convert them to interoperable types. Functions that use non-interoperable types are not refactored.

When a function is defined in one language and called by code written in another language, it is important that the argument and return types be compatible. Consider passing a `float` value from C to Fortran. Fortran's equivalent type is `real`, and in the case of Fortran 2003, `real(kind=C_FLOAT)`. If a programmer wants to change C/Fortran legacy code to follow the Fortran 2003 interoperability standard, she must use the correct data types in each context.

Sometimes the data types may be incompatible. For example, FORTRAN 77 and Fortran 90 allow a logical type to be of various kinds (distinguished by the `KIND` parameter). But Fortran 2003 only allows one interoperable logical type (`C_BOOL`, the `KIND` value implicitly defined to be 1) which corresponds to the `_Bool` datatype of C. If a program uses data types not supported by Fortran 2003, they cannot be refactored automatically.

Table 3 shows the data types of function parameters for the functions that communicate using ad hoc interoperability mechanisms (791 functions from Table 2). We analyzed the compatibility issue of the shared data types across different interoperability standards.

As expected, intrinsic data types (`INTEGER` and `REAL` types in Fortran) are shared most commonly. However, not all intrinsic types are interoperable, depending on the interoperability convention used. There were 50 `unsigned` data types[1] passed from C to Fortran via ad hoc interoperability mechanisms, but Fortran 2003 does not support the `unsigned` type. In these cases, the Fortran code treats the C `unsigned` data as a signed type. Consider the *mp_assign_to_cpu_* function (mp_assign_to_cpu.c, line 266) from G:

```
C parameter :    uint32_t *relative_cpu
Fortran data:    integer nowcpu
```

Here, Fortran assumes that the passed data is a signed integer. This is unsafe: it may lead to a signedness problem [4]—even a security vulnerability [21]—if the assumption is not right. The programmers have to ensure that the code is not dependent on the data type change. Some Fortran compilers support unsigned types (e.g., the Sun Studio Fortran 95 compiler [34]), but it is not widely adopted—we did not find

[1]This excludes `unsigned char` types. They are supported in Fortran 2003. However, there were no such instances in our corpus.

any instances of unsigned intrinsic types in the corpus.

Guided by the empirical data, our CONVERT TO STANDARD DATA TYPE (CSDT) refactoring (§ 6) operates in two modes. In the aggressive mode, the refactoring converts the unsigned type to a signed type on the Fortran side. In the safe mode, CSDT does not refactor the interface following ad hoc standard.

When a string (i.e., character pointer) data type is shared, typically extra parameters are (optionally) added at the end of the C function to denote the length of the strings that are passed from Fortran. Table 3 lists 145 string data types that were shared. In 133 instances, extra integer parameters denoting length are added to the interoperable C functions. Consider the extra `host_name_len` parameter in the *tcp_connect_* function from WWW, as shown in the motivating example (§ 2). In the remaining 12 instances, the string length was fixed since a string literal was passed; so it was not passed explicitly.

The extra length parameter creates an issue when parameter orders are changed in a CHANGE FUNCTION SIGNATURE (CFS) refactoring (§ 6)—the corresponding length parameters have to change as well for the reordered string parameters. However, Fortran 2003 removes the need for the extra parameter. A CFS refactoring design is more clean for code following this standard. We implemented CFS so that it only supports Fortran 2003-compliant code.

Array variables are frequently shared between C and Fortran (243 instances in Table 3). Most of the shared arrays contained basic data types. We found only 2 instances where an array of structs is passed. This makes the Fortran 2003 interface definition more complicated, since the details of the struct have to be included. The CONVERT TO STANDARD DATA TYPE (CSDT) refactoring (§ 6) supports arrays with complex data types.

Table 3 lists 76 pointer variables[2] that were shared. The Fortran convention is to create an `INTEGER` wide enough to hold the pointer passed from the C side, although there is a `pointer` attribute. In 70 instances, the pointer types from the C side were cast to integer data types. Only WWW has 6 instances that used the `pointer` attribute (it also uses the cast-to-integer approach in 3 instances). The two code snippets are from WWW, showing two different ways to share pointers.

```
C parameter :  void (**sub)()
Fortran datatype:  integer, intent(in) :: sub

C parameter :  void *calc_handle
Fortran datatype:  integer, pointer :: calc_handle(:)
```

The CONVERT TO STANDARD DATA TYPE (CSDT) refactoring supports functions that share a single pointer data type. 21 out of 40 single pointers listed in Table 3 are non-void pointers; they are converted to integers. 19 void pointers are converted to `Type(c_ptr)` and passed by value using the `value` attribute, according to Fortran 2003 standard. CSDT does not support function pointers, since there were few instances of those. However, this can be added easily using the `Type(c_funptr)` type introduced in Fortran 2003. Pointers with multiple indirection levels can be supported by using `Type(c_ptr)` and casting them to void pointers. However, this may be unsafe based on how memory is managed. Therefore, CSDT does not handle multiple pointers.

[2]This excludes character pointers (strings). Character pointers are listed separately in column 4 of Table 3.

Table 4: Parameter Passing Mechanism (Fortran Routines Calling C Functions)

App. ID	Passed by value	Passed by reference		
		Value defined in C	Intrinsic unchanged in C	Other unchanged in C
AST	0	32	23	13
CCSM3	0	22	11	2
CCSM4	0	21	10	0
CESM	55	19	7	0
CFS	0	5	4	0
CM	0	41	19	54
CMCC	0	6	0	5
G	0	17	5	0
MECH	1	1	0	0
MODEL	2	2	2	0
NWC	0	109	65	92
WRF	2	39	3	2
WWW	0	40	5	8
Total	60	354	154	176

There were 101 derived types (struct types) shared between C and Fortran in the applications in our corpus (Table 3). The CONVERT TO STANDARD DATA TYPE refactoring supports derived types using an explicit `BIND` attribute as introduced by the Fortran 2003 standard.

Finally, Table 3 lists 28 `logical` types in our corpus. None of them used a `KIND` parameter, so there were no compatibility issues converting them to follow the Fortran 2003 standard. The `logical` type instances in Fortran used an `int` type in C code to share data. The CONVERT TO STANDARD DATA TYPE (CSDT) refactoring converts them to use C's `_Bool` type.

5.3 RQ3. Parameter Passing Mechanism

How often are arguments passed by reference vs. by value? How often are arguments passed by reference when they could be passed by value instead?

Key Result: There are many instances that a variable is passed by reference, but it is sufficient to pass it by value. **Design Decision:** Use data flow analysis to find if a parameter is used in a read only manner; pass such parameters by value from Fortran to C. Pass other parameters by reference.

Fortran only supports call-by-reference, while C supports both call-by-value and call-by-reference. Thus, when a C function calls a Fortran routine, the parameters have to be passed by reference, while a Fortran routine can pass parameters to C following both styles. If a variable can be passed by value, passing it by reference violates the principle of least privilege [28].

Table 4 lists how parameters are passed from Fortran to call C functions. Only 60 parameters ($\approx 8\%$) were passed using the `VALUE` attribute in their declaration statement (i.e., passed by value), following the Fortran 2003 standard. An example is in the *c_field_list* function call in MODEL (system_tools.c): the `nl` parameter is passed by value.

```
C parameter  :  int nl

Fortran call :  integer (c_int), value :: nl
```

All applications using ad hoc interoperability mechanisms as well as some following the Fortran 2003 standard passed

all parameters by reference while calling C routines from Fortran. There were 684 instances of parameter passing by reference. Table 4 shows that 354 out of 684 (354/684 ≈ 52%) were modified inside the called C routine; therefore they ought to be passed by reference. About one-fourth of these parameters (154/684, ≈ 23%) were passed by reference but could have been passed by value. These are parameters with intrinsic types, e.g, integer, real, logical, and unsigned types. Consider *sys_sleep_* from WWW (sysdepc.c):

```
Fortran parameter :  integer, intent(in) :: secs

// C function :
  /* Suspend execution for a specified time */
  void sys_sleep(secs)
    int *secs;
  {
    sleep(*secs);
  }
```

Here, the C function suspends the execution for the specified time. The value of `secs` never gets changed. So, it can be passed by value.

In contrast, some of the character array and intrinsic array types were not modified inside the called C function. However, they are passed by reference (176/684, ≈ 26%).

Since there were many instances of parameters unnecessarily passed as reference, we performed a reaching definitions analysis on the parameter and its aliases and used this to properly pass a parameter as a value or a reference inside the Convert To Standard Data Type and Transform To Standard Interface refactorings.

6. REFACTORING TO FIX C/FORTRAN INTEROPERABILITY ISSUES

Since very few programs follow the ISO interoperability standard (RQ1, § 5), there is a need for an automated approach to assist programmers in updating to the new standard. We describe two refactorings (CSDT and TSI) for this purpose. We also describe a multi-language version of Change Function Signature to evolve C/Fortran interoperable functions.

6.1 Convert to Standard Data Type (CSDT)

The Convert To Standard Data Type refactoring converts data declarations to use standardized C-interoperable types introduced in Fortran 2003.

Motivation: To standardize interoperability, it is necessary that Fortran data shared with C functions be interoperable with a C type, as described in the ISO Fortran standard. The types that are not supported should be left alone. Without automated tooling, a programmer has to manually transform such interoperability instances.

Precondition: A programmer selects a C function name or definition and invokes the CSDT transformation. The following precondition is checked:

- The function is an interoperable one—it calls a Fortran subprogram or is called by a Fortran subprogram.

Mechanism: The refactoring modifies the types to follow the Fortran 2003 standard.

CSDT maintains a mapping between interoperable Fortran types and their equivalent C types; some strategies are described in Table 3. For each shared variable, CSDT determines the C type and the equivalent Fortran type and type parameter. It performs a data flow and dependence analysis on the C side to determine whether any parameter is passed by reference but is used in a read-only manner. If such a parameter is passed by reference from the Fortran code, CSDT converts the Fortran type declaration into its standard form. If the type is not interoperable with C, an error is raised.

Example: Consider the motivating example from § 2, extracted from WWW. The following shows the updated data types:

```
integer(C_INT):: sock
character(C_CHAR), intent(in):: host_name(*)
integer(C_INT), intent(in), value :: port
logical(C_BOOL), intent(out):: error
```

CSDT maps the data types of the parameters passed to follow the Fortran 2003 standard, e.g., `C_BOOL` is used to represent logical types. The data flow analysis identifies that the variable `port` is only used as read only value. CSDT suggests that it should be passed by value.

6.2 Transform to Standard Interface (TSI)

The Transform To Standard Interface refactoring introduces a Fortran `interface` block corresponding to a C function. The parameter and return type declarations use the C-interoperable types defined in the Fortran 2003 standard.

Motivation: Fortran 2003 provides syntax that makes the names and types of interoperable procedures explicit. This transformation ensures that the interface between C and Fortran follows this standard.

Precondition: A programmer selects a C function name or definition and invokes the TSI refactoring. The following preconditions are checked:

- The function is an interoperable one—it calls, or is called by a Fortran procedure.
- The selected interoperable function uses a legacy mechanism for interfacing.

Mechanism: The refactoring introduces a Fortran 2003-compliant `interface` block to explicitly describe the interoperable interface.

TSI first determines all the function call expressions both in C and in Fortran code that refer to the selected function. It removes the trailing underscore from the name of the function and all the function call expressions on the C side. On the Fortran side, TSI adds a reference to the `iso_c_binding` module and an `interface` block with appropriate `BIND(C)` declaration for the procedure, based on the C function prototype. TSI calls the CSDT refactoring to construct parameter declarations in the interface that is introduced. If Fortran passes any character string data to C and the C code adds an extra length parameter, the C function signature must be changed to drop that parameter and instead introduce a local variable inside the function to refer to the string length. Finally, if there is an `external` declaration for the selected C function in the Fortran code, TSI removes it.

Example: Consider the running example from § 2. The *tcp_connect_* function in the C program used the appended-underscore naming convention. TSI defines an interface on the Fortran side.

```
interface
  subroutine TCP_CONNECT(sock, host_name, port, error
      ) bind(C)
   use iso_c_binding
    integer(C_INT):: sock
    character(C_CHAR), intent(in):: host_name(*)
    integer(C_INT), intent(in), value :: port
    logical(C_BOOL), intent(out):: error
  end subroutine
end interface
...
...
subroutine client_connect(handle, host_name,
    host_port, error) bind(C)
    use iso_c_binding
    ...
    call tcp_connect(fd_table(handle)%fd%socket,
        host_name, host_port, error)
    ...
    return
end subroutine client_connect
```

The parameter types are determined by calling the CSDT refactoring.

On the C side, the trailing underscore for the function can be removed.

```
void tcp_connect (sock, host_name, port, error)
 int *sock, *port, *error;
 char *host_name;
 {
   int host_name_len = strlen(host_name);
   ...
 }
```

Also, the `host_name_len` parameter from the function definition has been removed from the C function signature, and a local variable has been introduced instead. This is because the Fortran compiler no longer passes hidden string length parameters.

6.3 Change Function Signature (CFS)

The CHANGE FUNCTION SIGNATURE refactoring allows a function's interface to be changed, with all call sites updated accordingly. A typical use is parameter reordering. For an interoperable function, this requires updating the function definition, declarations, and call sites in both C and Fortran code.

Motivation: After making an interoperable function compliant with the Fortran 2003 standard (perhaps using the previous two refactorings), there may be a need to evolve its interface in the future, e.g., swapping the parameter order. This affects both C and Fortran code.

Precondition: A programmer selects a C function name or definition and invokes the CFS refactoring. The following preconditions are checked:

- The function is an interoperable one—it calls a Fortran routine or any Fortran routine calls it.
- The selected interoperable function uses the standard interfacing mechanism by using a binding label match.

Mechanism: The refactoring modifies the interface definition on the Fortran side (if needed), the function definition on the C side, and call sites on both C and Fortran sides. Currently, the refactoring supports swapping parameter order only; other features such as adding and deleting parameters could be added in the future.

Example: Consider the running example from § 2, updated to follow the Fortran 2003 interoperability standard by applying the CSDT and TSI refactorings. Suppose a

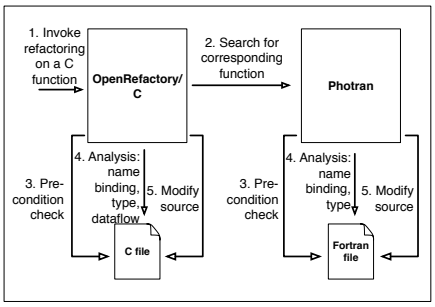

Figure 1: Refactoring Steps

programmer wants to apply CFS on this updated program to reorder the first two parameters of the *tcp_connect* function. The modified C code shows the parameters `sock` and `host_name` in swapped order.

```
void tcp_connect (host_name, sock, port, error)
 int *sock, *port, *error;
 char *host_name;
 {
  ...
 }
```

Similar changes are made on the Fortran side. The following shows the modified function call.

```
  call tcp_connect(host_name, fd_table(handle)%fd%
      socket, host_port, error)
```

7. EVALUATION

We prototyped the three refactorings by building on the C and Fortran refactoring infrastructures in OpenRefactory/C [13] and Photran [27], respectively. Both are Eclipse plug-ins; each has its own abstract syntax tree and its own mechanisms for resolving name bindings, variable types, etc. We created a new plug-in to automate the refactorings, dependent on both, that analyzes code in each language separately. When a user invokes a refactoring on a C function, the OpenRefactory/C module passes the information to the Photran module and searches for corresponding interoperable routines and the files that contain the routines. Then the refactoring engines perform the precondition checking, analyses, and transformations separately (Figure 1).

We tested the refactorings manually on our test corpus. We did not test all the interoperable function calls. We randomly selected the refactoring targets in all the programs. However, we covered some instances for each type of interoperable issue. Moreover, the code coverage of our refactoring shows that we cover a major part of the refactoring implementation while we were testing (90% statement coverage for the C side implementation and 60% statement coverage for the Fortran side implementation for TSI:BIND_ refactoring; 76% and 34% for the C and Fortran implementations for TSI:external refactoring). After the refactoring, we validate that our refactoring did not introduce syntactic errors. Table 5 shows the number of function calls that were tested for each of the programs. We also ran four programs (LP, MAG, RIM, and WRF) before and after applying TSI and CSDT refactorings (Table 5). LP, MAG, and RIM ran with the same input; for WRF, we ran the accompanying testsuite. The programs executed the refactored code segments and generated the same output before and after the refactorings. Other programs were not tested because they required special hardware or libraries to be present.

Table 5: Testing the Implemented Refactorings

Program	# of fn calls	# tested	CSDT		TSI: BIND_		TSI: external		CFS	
			# completed	# prevented	# completed	# prevented	# completed	# prevented	# completed	# prevented
AST	41	0	0	0	0	0	0	0	0	0
CCSM3	100	18	18	0	15	0	2	1	14	4
CCSM4	37	15	15	0	0	0	15	0	6	9
CESM	35	10	9	1	0	0	9	1	7	3
CFS	2	0	0	0	0	0	0	0	0	0
CM	13	9	9	0	9	0	0	0	8	1
CMCC	14	3	3	0	3	0	0	0	0	3
COM	2	0	0	0	0	0	0	0	0	0
ELE	3	0	0	0	0	0	0	0	0	0
G	8	8	7	1	6	1	1	0	5	3
LP	3	2	2	0	2	0	0	0	2	0
MAG	1	1	1	0	1	0	0	0	1	0
MODEL	1	1	1	0	1	0	0	0	1	0
NWC	500	28	28	0	12	1	14	1	20	8
RIM	3	2	2	0	2	0	0	0	2	0
SOL	1	1	1	0	1	0	0	0	1	0
WRF	12	5	1	4	1	4	0	0	0	5
WWW	18	8	8	0	6	0	2	0	6	2
YOD	2	1	1	0	1	0	0	0	1	0
Total	796	112	106	6	60	6	43	3	74	38

Not all function calls could be refactored. For example, when CSDT is applied to a function that has an unsigned integer parameter, the type is not cast to a signed integer to be safe [21]. The refactoring does not make any changes to the program; this is the correct behavior. Table 5 shows that when CSDT was applied to G, it allowed the refactoring to continue for 7 cases and prevented 1 case (`unsigned` parameter). Again, when we applied TSI on the same instances, it prevented the refactoring in 1 case because of the problem mentioned above. Finally, we applied CFS to our standardized interface, and it prevented the refactoring for 3 instances. This is because those functions had one or zero parameters; the parameters could not be swapped.

We expected a few other cases where refactoring was prevented. We did not support interoperability with macros (e.g., `F77_FUNC_`); attempts to refactor those procedures were blocked. Another common reason for blocking a refactoring was the presence of functions on the C side that would be identically named if not for the underscore, such as `foo_` and `foo`. Updating the former would create a naming conflict and, as such, were avoided. Note that preventing these refactorings during the precondition step is the correct behavior.

More details about the project are available online at https://sites.google.com/site/cfortranrefactoringtool/

8. THREATS TO VALIDITY

The main concern about the empirical data in this paper is whether the corpus is representative of interoperable C and Fortran programs at large. The authors of this paper have years of experience of working with Fortran programmers and provide refactoring tool support for them. Many applications in the corpus were derived from the corpus used to test the Photran [27] infrastructure, an Eclipse project that supports 40 Fortran-specific refactorings. Additionally, the programs represented different application domains and were written and/or evolved over a period of time so that they were exposed to several interoperability standards (§ 4.2). Most importantly, our intention was not to present the empirical data to reflect the general state of C and Fortran interoperability but to use the data as a case study to motivate the need for a refactoring-based solution and to guide the design decisions of these refactorings.

9. FUTURE WORK AND CONCLUSION

The area of cross-language analysis and refactoring has significant potential for future work. Most vendors for Fortran compilers also produce C compilers; type-checking Fortran procedure calls against declarations in C headers is feasible, at least in theory. Lint tools could identify problematic uses of data passed from another language (e.g., C code iterating through a Fortran array in row-major order). Checking and inferring `restrict` qualifiers for function parameters (e.g., [2]) is more complex in a multi-language setting. Pushing the idea of cross-language refactoring to its limits, one could pursue cross-language function extraction and inlining (recognizing, of course, that this could only be successful in limited scenarios).

The world is increasingly moving toward polyglot programming, and tools that support polyglot programs are inevitable. In this paper, we considered C/Fortran interoperability as a case study and presented empirical data exploring 20 applications totaling more than 12 million lines of code. Our results show that most of the programs still use legacy, compiler-dependent techniques to achieve interoperability. While interoperability was incorporated into the Fortran 2003 standard, it may have come too late—many programs never adapted to the new standard. Automated refactoring tools could prove to be a solution to this problem; we prototyped three cross-language refactorings, demonstrating that automated refactoring may be a viable means to help programmers eliminate the use of legacy interoperability mechanisms from their code in the future.

10. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1217271. The authors thank the anonymous referees for their helpful feedback.

11. REFERENCES

[1] J. C. Adams, W. S. Brainerd, R. A. Hendrickson, R. E. Maine, J. T. Martin, and B. T. Smith. *The Fortran 2003 handbook: the complete syntax, features and procedures*. Springer Science & Business Media, 2008.

[2] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 129–140, New York, NY, USA, 2003. ACM.

[3] J. Bielman, L. Oliveira, D. Knapp, E. Backes, and S. Compall. *CFFI User Manual*, 2006.

[4] blexim. Basic integer overflows. *Phrack*, 60, 2002.

[5] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In *Future of Software Engineering, 2007. FOSE '07*, pages 326–341, May 2007.

[6] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. P. Jones, A. Reid, M. Wallace, and M. Weber. *The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report*, 2003.

[7] N. Chen and R. Johnson. Toward refactoring in a polyglot world: Extending automated refactoring support across java and xml. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 4:1–4:4, New York, NY, USA, 2008. ACM.

[8] I. D. Chivers and J. Sleightholme. Compiler support for the Fortran 2003 standard. *ACM Fortran Forum*, 26(1):7–9, Apr. 2007.

[9] B. Cossette and R. J. Walker. DSketch: Lightweight, adaptable dependency analysis. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 297–306, New York, NY, USA, 2010. ACM.

[10] L. Frenzel. The language toolkit: An API for automated refactorings in Eclipse-based IDEs. *Eclipse Magazine*, 5, jan 2006.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[12] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 321–336, New York, NY, USA, 2007. ACM.

[13] M. Hafiz, J. Overbey, F. Behrang, and J. Hall. OpenRefactory/C: An infrastructure for building correct and complex C transformations. In *Proceedings of the 6th Workshop on Refactoring Tools, Co-located with OOPSLA 2013. Indianapolis, Indiana*. ACM, 2013.

[14] Joint Technical Committee ISO/IEC/JTC1. *ISO/IEC Standard 1539-1: Fortran 03*, 2004.

[15] M. Kempf, R. Kleeb, M. Klenk, and P. Sommerlad. Cross language refactoring for eclipse plug-ins. In *Proceedings of the 2Nd Workshop on Refactoring Tools*, WRT '08, pages 1:1–1:4, New York, NY, USA, 2008. ACM.

[16] S. Li and G. Tan. Finding bugs in exceptional situations of jni programs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 442–452, New York, NY, USA, 2009. ACM.

[17] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[18] P. Linos. PolyCARE: A tool for re-engineering multi-language program integrations. In *Engineering of Complex Computer Systems, 1995. Held jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP WRTP, Proceedings., First IEEE International Conference on*, pages 338–341, Nov 1995.

[19] P. Linos, Z. hong Chen, S. Berrier, and B. O'Rourke. A tool for understanding multi-language program dependencies. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 64–72, May 2003.

[20] P. Linos, W. Lucas, S. Myers, and E. Maier. A metrics tool for multi-language software. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, SEA '07, pages 324–329, Anaheim, CA, USA, 2007. ACTA Press.

[21] Magma /FHC. Exploiting memory corruptions in Fortran programs under UNIX/VMS. *Phrack*, 67, 2010.

[22] P. Mayer and A. Schroeder. Cross-language code analysis and refactoring. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 94–103, Washington, DC, USA, 2012. IEEE Computer Society.

[23] M. Méndez, F. G. Tinetti, and J. L. Overbey. Climate models: challenges for fortran development tools. In *Proceedings of the 2nd International workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*, pages 6–12. IEEE Press, 2014.

[24] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22, Sept 2005.

[25] D. Moise and K. Wong. Extracting and representing cross-language dependencies in diverse software systems. In *Reverse Engineering, 12th Working Conference on*, pages 10 pp.–, Nov 2005.

[26] National Oceanic and Atmospheric Administration (NOAA). The World Magnetic Model, 2010.

[27] Photran - An Integrated Development Environment and Refactoring Tool for Fortran. http://www.eclipse.org/photran/.

[28] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep 1975.

[29] H. Schink. sql-schema-comparer: Support of multi-language refactoring with relational databases. In *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, pages 173–178, Sept 2013.

[30] H. Schink, M. Kuhlemann, G. Saake, and R. Lämmel. Hurdles in multilanguage refactoring of hibernate applications. In *In Proceedings of the 6th International Conference on Software and Database Technologies, pages 129 - 134. SciTePress -Science and and*

Technology Publications, 2011.

[31] P. S. Shenkin. Writing c functions callable by fortran and vice versa. In *ACM SIGPLAN Fortran Forum*, volume 18, pages 6–12. ACM, 1999.

[32] D. Strein, H. Kratz, and W. Lowe. Cross-language program analysis and refactoring. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 207–216, Washington, DC, USA, 2006. IEEE Computer Society.

[33] D. Strein, R. Lincke, J. Lundberg, and W. Lowe. An extensible meta-model for program analysis. *Software Engineering, IEEE Transactions on*, 33(9):592–607, Sept 2007.

[34] Sun Microsystems, Inc. *Fortran User's Guide*, 2004. Sun Studio Fortran 95 compiler, `f95`, 819-0492-10.

[35] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 365–377, Berkeley, CA, USA, 2008. USENIX Association.

[36] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154–164, 2000.

[37] F. Tomassetti and M. Torchiano. An empirical assessment of polyglot-ism in GitHub. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 17:1–17:4, New York, NY, USA, 2014. ACM.

[38] F. Tomassetti, M. Torchiano, and A. Vetro. Classification of language interactions. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on*, pages 287–290, Oct 2013.

[39] A. Vetro', F. Tomassetti, M. Torchiano, and M. Morisio. Language interaction and quality issues: An exploratory study. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 319–322, New York, NY, USA, 2012. ACM.

[40] World Climate Research Programme. CMIP—Coupled Model Intercomparison Project, 1995.