Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

The Design and Evaluation of an Interoperable Translation System for Object-Oriented Software Reuse

Stephen Schaub[a] Brian A. Malloy[a]

a. Clemson University, Clemson, South Carolina, USA

Abstract

In this paper, we address the problem of defining a source-to-source translation system for reusable software components. We design an interoperable language for writing software components, and present a system to translate components written in the interoperable language to a set of compatible target languages. We analyze the common features in a set of popular programming languages to inform the design of our interoperable language. We evaluate the utility of our system by using our source-to-source translator to convert two well-known open source Java libraries to C++ and Python, and assess the accuracy and performance of the resulting translation.

Keywords Source-to-Source Translation, Programming Language Design, Language Interoperability, Software Reuse.

1 Introduction

The software development landscape has changed radically in its relatively short history. The introduction of portable high-level languages, and in particular, the ascendance of object-oriented technologies, has enabled a degree of software reuse that was unimaginable to the assembly language programmers of the 1950's. And yet, solutions to fundamental problems in software engineering reuse and reliability remain elusive and in many cases unsolved. The proliferation of high-level languages—the mechanism enabling reuse—has been a significant factor in fueling this growth but has also increased the scope of the problem, thereby retarding progress in arriving at solutions.

One approach to interlingual code reuse involves the use of tools that translate the source code of complete programs or libraries from one language to another. Although commercial systems are available for this purpose, they generally suffer from a number of limitations that constrain their usefulness. Terekhov and Verhoef observe that the problem of completely automated source-to-source translation is a difficult one, and that large language conversion efforts are a risky undertaking that have led to bankruptcies and dismantled departments [TV00]. In this work, we propose solutions to some of these challenges, and demonstrate their utility through the implementation and validation of a system that implements these solutions.

Another approach to building a fully automated source-to-source translation system involves defining subsets of languages between which programs can be automatically translated [AGG+80, Pla13, AGLM14]. We believe that the definition of such a subset should be guided by an analysis of a set of popular, high-level languages that share a significant set of semantically compatible features. We assert that an interoperable language whose design is constrained by these compatible features will be rich enough to enable the implementation of useful software components, and that components written in such a language can be automatically translated to compatible languages with good reliability. We acknowledge at the outset that some aspects of languages are commonly left unspecified in language definitions. When questions like these are left to the language implementation to resolve, no platform- and implementation-independent translation system, automated or otherwise, can hope to achieve complete semantic equivalence. But this limitation should not preclude the exploration of the practical utility of such systems.

Currently, there is no effort described in the literature that facilitates the design and implementation of a source-to-source translation system for enabling software component interoperability that is based upon a carefully specified concrete subset of a high-level language. The existence of such a system would enable the creation of libraries of components that can be translated for use in any of the supported target languages.

In this paper, we have created a system for translating components written in an interoperable language, *iJava*, into a set of compatible target languages. We demonstrate the utility of our approach by translating components written in Java to Python and C++. To facilitate the translation, the system utilizes a graphical representation of the interoperable component source code. Like the interoperable language, this representation is language-neutral, devoid of any target-language-specific details. The generality of this representation enables a modular architectural approach that facilitates the enhancement of the system to support additional, compatible target languages. The set of compatible target languages would be those for which iJava features either map directly to native structures, or can be emulated with native structures. The use of a graphical representation also enables the use of graphical transformations to perform the code generation.

Contributions. Our main contributions are as follows:

- An analysis of a set of object-oriented languages identifying common language features. No comprehensive analyses of this type has been conducted to date.
- A language, and corresponding abstract semantic graph, designed for straightforward translation to the analyzed languages.
- The design and implementation of a tool that translates code from Java to C++ and Python based on our approach.

2 Related Work

In this section we survey the field of language interoperability and present alternatives to source-to-source translation. We then discuss various approaches to source-to-source translation systems.

2.1 Language Interoperability

Language interoperability involves the creation of programs that are written in more than one language. According to Weiser et al. [WDH89], language interoperability typically involves both low-level infrastructure issues such as the sharing of threads, I/O, and a common address space by code in multiple languages; as well as higher-level concerns, such as the ability of languages to invoke each others' subroutines and refer to each others' public identifiers, and the ability to share data representations. A variety of approaches to language interoperability have been explored in the literature. In this section, we present a survey of the approaches.

In one approach to language interoperability, clients in language A access components in language B via *remote procedure calls* (RPC). This model enables a distributed computation, as there is no sharing of address space between the language environments that comprise the overall program. Weiser et al. argue that this approach can work well when the language partitioning logically mirrors the client-server model, although the overhead of remote procedure calls can introduce a significant cost [WDH89].

Another approach to interoperability involves the use of *foreign language interfaces*, which provide a mapping between the calling conventions of language A and language B. Such mappings allow programs in language A to invoke subroutines in language B. This approach is used by tools such as SWIG [Swi15], which automatically generates interfaces for a variety of languages to be able to consume components written in C and C++. This technology is more efficient than the RPC approach, since code for both languages typically resides in the same address space, but some overhead is still to be expected when traversing language boundaries; also, non-idiomatic interfaces can create a readability problem.

A third interoperability technique involves the use of a *common intermediate representation*. For example, Microsoft's .NET Framework [Pro02] and the Java Virtual Machine [LY99] both define an intermediate representation (IR) that supports several source languages. However, these approaches do not support all languages. For example, neither the Microsoft nor the Java IR supports multiple inheritance, thus excluding languages such as C++ that require it. In a variation of this approach, Weiser et al. describe a portable common runtime that acts as a layer between the operating system and language-specific runtime systems, providing features such as memory management and threading support [WDH89]. This approach imposes fewer constraints on the interoperating languages than having a common IR, but has not seen widespread adoption.

Two languages are made interoperable through *source-to-source translation* when a module written in Language A is translated to source code in Language B, so that it can be utilized by a program in Language B. After the module is translated to Language B, it can be compiled or executed via interpretation in conjunction with the main program written in Language B.

The source-to-source approach to interoperability offers a number of advantages over the others we have described. It can offer greater efficiency, since there is no overhead due to interprocess calls or crossing foreign language interface boundaries, and no need to create a common intermediate representation that necessarily de-emphasizes efficiency to achieve interoperability. If the resulting code preserves good readability, after the component is translated, there is no need to preserve the original in order to maintain the component, yielding improved maintainability. Also, there is increased convenience, as there is no need to maintain two language environments to maintain both the component and the program that uses it.

2.2 Source-to-Source Translation

In this section we review the research that relates to source-to-source translation of languages for enabling software component interoperability. We first review research that establishes the usefulness of translating subsets of languages to other languages. We then review approaches to translating from one language to another language. We also review the literature that describes research for translating from one language to multiple languages, and finally we describe an approach to translating from multiple source languages to multiple target languages.

Source-to-source translation is a well established technique for language interoperability and software reuse. Citing Huijsman et al. [HvKPT87], Plaisted observes that previous work has shown the value of defining translations on subsets of languages [Pla13]. He argues for the development of abstract languages and subsets of high-level languages, and the creation of translators from those interoperable languages to other high-level languages. Rather than applying translation to large, special-purpose programs, he suggests source-to-source translation is more useful when applied to reusable algorithms, and urges the development of libraries of algorithms in these language subsets.

Many source-to-source systems have been developed that take a single source language as input and translate to a single target language. These translators are special-purpose conversion efforts that are not extensible to support other sources or targets. The literature contains numerous examples [HvKPT87, KLV03, MW91, TO98]. Most of these report challenges in performing a completely automated conversion of the entire source language. For example, Trudel et al. created a comprehensive Java to Eiffel translator that successfully ported some large test cases [TOFN11]. However, they were unable to translate aspects of Java that are unsupported in Eiffel, such as dynamic loading and weak references.

Relatively few source-to-source systems support multiple output languages generated from a common intermediate representation. Plaisted presents an abstract language, PL, that allows the construction of programs and proofs of correctness, and is designed to support the translation of programs into multiple concrete languages [Pla05]. However, he does not provide an implementation. Farrow and Yellin outline an approach for a many-to-many source-to-source translation system, and describe their experience implementing a bidirectional translation between Pascal and C [FY86]. Similar to our system, the authors design a common intermediate representation for the languages among which they wish to translate, guided in their design by analyzing the similarities and differences in the source and target languages. To perform the translation, the authors devise invertible attribute grammars, which allow them to specify the mapping from Language A to Language B, and then generate the inverse mapping from Language B to Language A automatically. They acknowledge that their system is unable to handle constructs that cannot cleanly map between both languages. They do not evaluate their system as to its completeness or effectiveness.

Albrecht et al. describe a similar effort to translate between subsets of Pascal and Ada [AGG+80]. The authors performed an analysis of Pascal and Ada, defining a common subset of both languages using an intersection analysis similar to our approach. They devised a common tree structure capable of representing programs written in the compatible subsets of Pascal and Ada, and a tool set for translating Pascal and Ada programs into the graphical form, and back to Pascal and Ada. The authors observed that, even with languages as closely related as Pascal and Ada, there are subtle semantic differences that create difficulties for defining compatible sublanguages. We view our work as expanding their approach to a larger set of languages with a richer set of features.

Arrighi et al. describe a recent effort to produce a many-to-many translation system, GOOL [AGLM14]. Their system is a source-to-source translator for object-oriented languages. The translator supports multiple high-level input and output languages, mapping input programs to an intermediate representation based on an abstract object-oriented language, GOOL, and generating output to the target high-level language from that intermediate representation. The GOOL system is designed to handle a subset of features of its input languages; when it encounters programs that use unsupported features, it passes through the unsupported fragments as comments in the output program. Thus, it is intended as a human-assisted translation system, requiring manual intervention to handle unsupported features, rather than a fully automated system. No language specification is available for the abstract GOOL language, so we are unable to evaluate the range of functionality their system can support. Also, the authors have not published any reports validating their system, and in our tests, the GOOL system had difficulty handling fundamental constructs such as basic string operations.

We distinguish our work from GOOL in the following ways. First, although both systems make use of an intermediate language that represents a subset of functionality available on popular object-oriented languages, our language provides a specification that defines the supported features. Second, our system supports the fully automated translation of programs that use features not directly supported by the intermediate language, rather than passing them through as comments to be handled by a manual translation step. Finally, our system explicitly supports the translation of unit tests to be used in validation of the translated code. We note that GOOL claims to support multiple high-level input languages, while we are proposing to support only one high-level input language. However, there is nothing about our approach that prevents us from supporting multiple high-level input languages.

The rest of this paper is organized as follows. In the next section we describe the important challenges involved in building a source-to-source translator and an overview of the solutions that we provide in this paper. In Section 4 we provide an overview of the design and implementation of our system and, in Section 5, we describe the design of our intermediate language, *iJava*, or *interoperable Java*. In Section 6, we review the implementation of our translation system for translating from our source language, *Java*, to the target languages, *C++* and *Python*. In Section 7, we evaluate the quality of our translation process and, finally, in Section 8 we summarize our findings and discuss future work.

3 Challenges

In this section, we discuss source-to-source translation, its challenges, and some proposed solutions.

3.1 Source-to-Source Challenges

Designers of source-to-source translators have several goals. An effective source-to-source translator must accurately preserve the semantics of the input program, so that the translated version produces the same results as the original when executed. A comprehensive translator will map all constructs in the source language to the target, yielding a fully automated translation. Efficiency of the generated code is also an important consideration. Finally, a good translator will produce a translated program that is readable, ideally yielding highly maintainable code that is idiomatically expressed in the target language.

A number of challenges are involved in applying source-to-source translation to the problem of achieving component interoperability. When source language features are used in the component that are not present in the target language, those features must be emulated in the target; depending on the difficulty of the emulation, the readability and efficiency of the resulting code can suffer greatly. For example, mapping unsafe type operations in languages like Fortran to strongly typed languages like Ada, which prohibit such operations, can be done by modeling the entire memory store using a vector [MW91]—yielding singularly unsatisfying results.

Even when a particular feature is available in both languages, semantic differences and undefined behaviors can complicate translation. As an example of the difficulties arising from semantic differences, consider integer overflow. In the C family of languages, integer overflow results in rollover, a behavior on which some programs rely for correct operation. However, in In Ada and Visual Basic, integer overflow triggers an exception. As for undefined behaviors, C and C++ leave numeric data type sizes as an implementation detail, and do not specify the behavior that occurs when array bounds are violated. When such issues are left unspecified by a language specification, producing a reliable translation that is platform- and implementation-independent is impossible.

Finally, translation of standard library calls poses a significant challenge. Ideally, a source-to-source translation should map standard library calls for the source language to the native standard library calls for the target language, rather than introducing an interoperable library layer. However, since standard libraries differ widely in the range of features provided, such a mapping is not possible, in general. For example, standard library facilities used in the source may not be available in the target language, and even when they are, their semantics may differ substantially. In cases when the library source code is unavailable, a completely automated source-to-source translation is impossible.

3.2 Addressing Source-to-Source Challenges

One approach to addressing these issues is to limit the scope of the problem, by restricting source programs to using a subset of language features that form an intersection of the desired target languages. Although this is not a practical solution for tackling the conversion of much legacy code, there are important software reuse scenarios that could be enabled by such a system. As Plaisted observes, general-purpose algorithms such as those used to process data structures have characteristics that differ from code found in custom application programs: they contain relatively few calls to the standard library; tend to be more carefully specified and written; venture less frequently into corner areas of the language that create difficulties for translation; and are frequently reused [Pla13]. Plaisted envisions creating libraries of reusable, general-

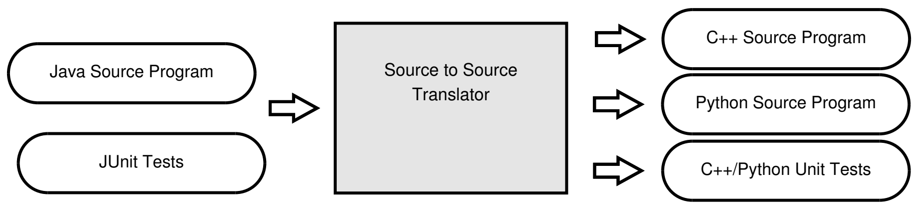

Figure 1 – System Overview

purpose software components, written in an interoperable subset language, that are designed to be automatically translated to compatible target languages. And, as we demonstrate, such an intersection is rich enough to enable the conversion of legacy libraries that have few standard library dependencies.

We address the problem of translating features in the source language for which there are no equivalent features in the target by careful design of the source language. We restrict the set of features in the source language to those that can be mapped in a straightforward manner to all of the selected target languages.

To address the problem of semantic differences between languages, we utilize two strategies. In some cases, we leave some behaviors unspecified in our interoperable language. For example, since the behavior of integer overflow differs among our languages, our interoperable language will leave this behavior undefined. In other cases, we will rely on libraries to bridge the semantic gap. For example, our interoperable language will specify automatic memory management semantics; in order to translate programs to target languages that do not support automatic memory management, we will rely on a garbage collection library to supply the needed functionality.

We address the problem of undefined behaviors by ensuring that any behaviors not defined in our target languages are also not defined in our source language. This will ensure that programs cannot rely on a given behavior in the source language that is undefined in the target.

To handle difficulties arising from standard library incompatibility, we define a minimal standard library for our interoperable language. We provide support libraries for our target languages that implement library facilities needed by our test cases that are not present in the target language.

4 System Overview

In this section, we present an overview of our source-to-source translation system. Figure 1 depicts a high-level view of the translator, which takes as input components, programs, and unit tests written in a subset of Java, and translates them to a set of supported target languages. To allow the validation of our system using test cases that do not conform to the iJava subset of Java, our translator accepts a subset of Java larger than iJava, internally transforming program structures not in the iJava subset into iJava-supported features.

The translator architecture is partitioned into stages to facilitate extensibility to additional target languages, as shown in Figure 2. The first stage converts the input Java program to an abstract representation of Java: an abstract semantic graph (ASG). For this stage, we utilize the high-quality Java parser component in the standard Java compiler, assuring good compatibility with a wide range of Java source

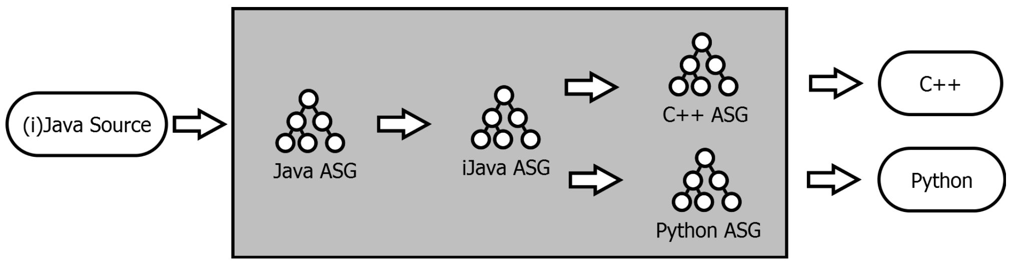

Figure 2 – Expanded System Overview

programs.

Stage 2 of our system transforms the program encoded in the Java ASG to a language-independent representation encoded as a iJava ASG. Since iJava is a subset of Java, this involves a set of graphical transformations, described in section 6. In particular, program structures utilizing Java features not present in iJava, such as for loops and anonymous classes, are transformed to structures that utilize iJava features.

In Stage 3 of our system, the iJava ASG is transformed to an ASG in the target language. This involves construction of a graph that closely mirrors the iJava ASG structure, but whose nodes contain data and logic needed for the generation of target code in the final step. The use of a target language ASG permits the use of tree transformations to implement some of the translation, and simplifies the final code generation stage.

In the final stage, target code is emitted during a traversal of the target language ASG. During this stage, we use a template-based code generation strategy.

5 Interoperable Language Design

In this section, we describe the design of our interoperable language, iJava. We begin in subsection 5.1 with a presentation of our analysis and design methodology. subsection 5.2 discusses how we selected a set of languages for analysis. Finally, we present the details of our analysis and language design in subsection 5.3.

5.1 Design Methodology

In order to determine a set of language features useful for defining an interoperable language, we analyze the intersection of the capabilities of a set of languages. The features included in this intersection determine the definition of our interoperable language, yielding features that can be mapped to suitable capabilities in the analyzed language set.

We agree with Plaisted's observation that interoperable component languages should be imperative languages with side effects, arrays, and records, because of the need for efficiency and applicability [Pla13]. To select the languages, we use the TIOBE index [TIO15], a well-known ranking of programming language popularity based on information provided by web search engines. To increase applicability of our results, we include both statically and dynamically typed languages. We exclude special purpose languages such as those designed primarily for database query and mathematical calculations, because they are not general purpose languages. We also exclude functional languages because of our focus on imperative language capabilities.

Finally, since the majority of languages in the top 20 TIOBE include object-oriented features, and including non-OO languages would necessarily exclude important features from our analysis, we narrow our focus to object-oriented languages.

After choosing the languages that we wish to consider, we compare the data types, operators, statements, and structural features of the languages. For a given language feature, we determine the intersection of the capabilities of that feature that are present in each language in our language set. This intersection represents a common subset of functionality for that feature that is found in all of the languages.

In some cases, an intersection of language features may exclude useful or important functionality that can easily be emulated in the languages where it is missing. We adjust the intersection to include the functionality in cases where the emulation does not significantly compromise the readability of the resulting code. For example, some languages include both a character sequence type (ex. Java's `String`) and a single character type (ex. Java's `char`), while others omit the single character type. A pure intersection would exclude the single character type, but since having a single character type is a useful feature, and can be easily emulated with a character sequence type, we include both types in our intersection.

In other cases, an intersection may yield undefined semantics. For example, the behavior of division by zero varies among the languages in our set, yielding an empty intersection. Rather than leaving the behavior of division by zero undefined, we define it, because the behavior can be easily implemented using a library function in languages where its behavior is different from our definition. In some situations, imposing a definition is not practical, and we must leave the semantics undefined.

As an example of our analysis methodology, consider the question of variable typing. Java's variables are statically typed; Python's are dynamically typed. To resolve this dichotomy, we consider which of these two approaches is *more restrictive*. Since statically typed variables are more restrictive than dynamically typed variables, we determine the intersection in this case to be statically typed variables. We observe that mapping programs from a statically typed language to a dynamically typed language requires only the erasure of explicit types from the variable declarations. Mapping code from a dynamically typed language to a statically typed language is difficult or impossible in cases where a variable holds more than one type of value over its lifetime, and, even when possible, leads to obtuse translations.

5.2 Analysis Set Selection

We studied languages in the February 2015 TIOBE top 20 (see Figure 3), and selected the following five languages for analysis, following the criteria discussed in Section 5.1:

- **C++**. An important object-oriented systems language. Consistently in the top 5 in the TIOBE index. The referenced language standard is C++14 [Int14].
- **Java**. A widely used object-oriented systems language. Java and C compete for the number one spot in TIOBE; both are consistently in the top 2. The referenced standard is Java 8 [GJS$^+$15].
- **Python**. A popular object-oriented scripting language. Twice language of the year in the TIOBE index. The referenced standard is Python 3.4 [pyt15].
- **JavaScript**. Known best for its use in client-side processing for web applications, JavaScript is an object-oriented scripting language consistently found

in the top 15 in the TIOBE index. JavaScript is based on an ECMA standard language named ECMAScript; the referenced standard is ECMAScript 5.1 [ECM15].

- **Object Pascal**. An object-oriented extension to Pascal, one of the most influential academic procedural languages of the late twentieth century. Object Pascal is consistently in the top 20 in the TIOBE index and one of the few that is not in the C family. In contrast to the other languages, there is no official language standard. This research is based on the language implementation Free Pascal, version 2.6.4 [pas15].

We selected these five languages in preference to others in the TIOBE top 20 because, within our overall target focus of general-purpose, imperative, object-oriented languages, we wanted a reasonably diverse mix of syntax, semantics, and execution model. The more diverse the set, the more tightly focused the intersection of language features, thereby increasing the size of the set of languages that would be potentially compatible with our interoperable language. These factors led us to exclude C, due to its lack of object-oriented features. Also, to increase diversity, we included Object Pascal and Python in preference to other curly-brace languages that were nearer the top of the list.

5.3 Language Analysis and Design

Rather than design a completely new language, we chose a subset of an existing language, Java, as the basis for our interoperable language. As our analysis shows, in most cases, a subset of Java's features often falls neatly into the intersection of our selected languages' features. This makes it a natural candidate for being the foundation of an interoperable language. Using a subset of an existing language has the practical benefit of eliminating the need for developers to learn a new language.

For our interoperable component language, we therefore define a subset of Java dubbed *iJava*: interoperable Java. A detailed language specification is available for Java, and its syntax and semantics are familiar to a large portion of the community. Rather than present a formal language specification for iJava, we define iJava in terms of how it departs from Java. Unless otherwise stated, iJava's features share Java's syntax and semantics.

The following subsections discuss iJava's data types, expressions and operations, statements, subroutines, organizational structures, exception processing, and standard libraries.

5.3.1 Data Types and Variables

In this section, we define iJava's features related to data types and variable definitions, and discuss related concerns. Table 1 presents a summary of the relevant language features in each of our surveyed languages, and shows which we chose to include in iJava.

Static vs. Dynamic Typing: As discussed in Section 5.1, iJava uses static typing.

Boolean type: Like all of our surveyed languages, iJava includes a special-purpose boolean type and associated logical and bitwise operations and, or, and not. (Although Object Pascal omits the logical boolean operators, its bitwise boolean operations act as logical operations when applied to boolean operands.) All of our

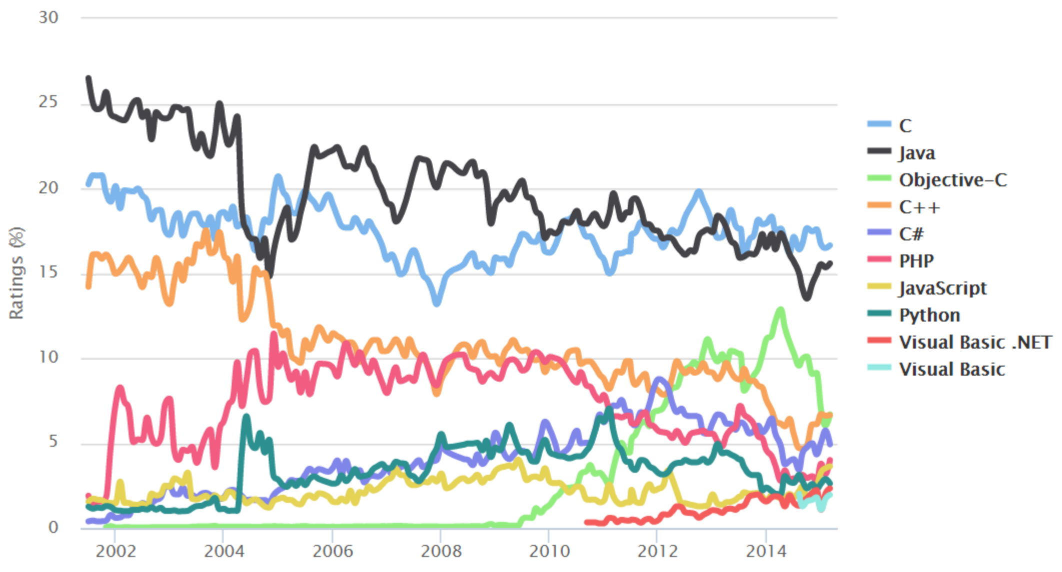

Figure 3 – TIOBE Index [TIO15]

languages perform conditional evaluation by default on logical boolean operations, so iJava adopts this behavior.

Numeric types: Our language analysis identified a common subset of numeric types including signed integers with storage sizes up to 64-bits, and a double-precision floating point type. (JavaScript has only a 64-bit floating point type, but available libraries can represent signed 64-bit integers using this type.) iJava adopts Java's byte, short, int, long, and double types.

Character and String types: iJava adopts Java's char and String types, following the logic discussed in Section 5.1.

Array types: iJava adopts Java's array functionality, including both single- and multi-dimension structures. This represents the results of our language intersection analysis that evaluated the diverse characteristics exhibited by the expandable and flexible Python list and JavaScript array, as well as the traditional statically typed arrays of Java, C++, and Object Pascal. Java's array semantics can be efficiently implemented in all of our surveyed languages.

Function Types: All of the analyzed languages allow functions (or pointers to functions) to be stored in variables and passed to and from subroutines. However, Java does not provide a special purpose function type, instead using interfaces to achieve an effective superset of this capability. Because Java provides no special purpose function type, we choose not to include one in iJava.

Reference and value types: As in Java, iJava's primitive types are value types, while its class and array types are reference types. This is compatible with Python and JavaScript, since the numeric and boolean values in these reference-type languages are immutable. It is also compatible with translation to C++ and Object Pascal.

Memory Management: Related to the issue of reference and value types is the question of memory management. All of our survey languages except C++ can track references to allocated memory and automatically deallocate values that become inaccessible. Since garbage collection libraries are available for C++, we choose to

	C++	Java	Object Pascal	Python	JavaScript	iJava
Typing						
Static vs. Dynamic	Static	Static	Static	Dynamic	Dynamic	Static
Explicit vs. Implicit	Explicit, Implicit	Explicit	Explicit	N/A	N/A	Explicit
Numeric Types						
Integer Precision	8–64 bit	8–64 bit	8–64 bit	arbitrary	N/A	8–64 bit
Unsigned Ints	Y	N	Y	N	N	N
Floating Point	single, double	single, double	single, double	double	double	double
Character Type	Y	Y	Y	N	N	Y
String Type						
Longest String	undefined	$2^{31} - 1$	unbounded	undefined	undefined	undefined
Mutable?	Y	N	Y	N	N	N
Boolean Type	Y	Y	Y	Y	Y	Y
Arrays						
Memory Alloc	Static, Dynamic	Dynamic	Static, Dynamic	Dynamic	Dynamic	Dynamic
Lower Bound	Implicit	Implicit	Explicit	Implicit	Implicit	Implicit
Records	Y	Y (class)	Y	N	N	Y
Reference / Value Types						
Primitives	Ref, Value	Value	Ref, Value	Value	Value	Value
Arrays	Ref	Ref	Ref, Value	Ref	Ref	Ref
Classes	Ref, Value	Ref	Ref, Value	Ref	Ref	Ref
Function Types	Y	N	Y	Y	Y	N
Garbage Collection	Library	Y	Y (COM based)	Y	Y	Y
Class Constants	Static, Instance	Static, Instance	Static	N	N	Static
Enumerations	Y	Y	Y	N	N	N

Table 1 – Data Type Analysis

include garbage collection in our subset of common features.

Variable declaration: Although Python has no variable declaration statement, and JavaScript's var statement is optional in some cases, a variable declaration statement is required by all statically typed languages and must therefore be included in our language subset. Of our statically typed languages, only C++ allows implicitly typed variable declarations; Java and Object Pascal both require the variable type to be specified in the declaration, so iJava requires explicitly typed variable declarations.

Constants: All of our statically typed languages provide mechanisms to define named constants in the context of a class definition. C++ and Java allow both static and instance constant members, while Object Pascal allows only static constants. Even though JavaScript and Python do not provide mechanisms to define constants, since this is an important feature in static languages, and constants can be emulated using variables and conventions, we choose to support static class constants in iJava.

Enumerated types: All of our statically typed languages provide a mechanism to define an enumerated data type, while our dynamically typed languages do not provide such a mechanism. We exclude enumerated types from iJava.

5.3.2 Operators and Expressions

In this section, we explore issues involving operators and expression evaluation.

Numeric operations: As shown in Table 2, iJava's numeric operations include the standard addition, subtraction, multiplication, division, and modulus operators; numeric cast conversion operators; bit shift; and comparison operators, all using standard Java semantics. This represents the broad consensus exhibited by our language set in this area. There is some variation in the precise semantics of the bit shift operators, but we choose Java semantics for iJava and handle semantic differences in other languages by the use of library routines.

Java specifies the behavior of arithmetic operations that result in numeric overflow and underflow, but iJava leaves these behaviors undefined, since there is variation among the surveyed languages in how these errors are handled. In the case of floating-point division by zero, where there is also disagreement among our surveyed languages, we choose Java semantics for iJava, because its behavior can be easily emulated in languages which handle this situation differently from Java.

String operations: iJava's string operations are provided by the java.lang.String class in its standard library, which includes a subset of the Java String methods, as described in Section 5.3.8. Also, Java's overloaded use of the + operator to perform string concatenation is supported in iJava.

Pointer expressions: C++ and Object Pascal permit the use of pointer arithmetic to allow a program to address arbitrary memory locations, provide an operation to take the address of arbitrary variables, and allow conversions between pointers and integers. Java, Python, and JavaScript do not provide such operations. Thus, the only pointer-related operations in iJava include the dereference operator, and equality/inequality comparison. We note in passing that because the result of dereferencing a null reference is undefined in C++, this operation is undefined in iJava.

Array expressions: We consider two array operations supported by all of our languages: length determination and indexing. All of our languages except C++ provide an operation to obtain the length of a dynamically constructed array. Since C++ provides array-like containers in its standard library that do provide this operation, we include this capability in iJava.

Array index bounds are checked in Java, JavaScript, Python, and optionally, in Object Pascal (using a compiler switch). Out-of-bounds accesses either trigger a runtime exception, or in the case of JavaScript, the value undefined. C++ does not perform bounds checking on its native arrays, but its standard library includes array-like containers that provide this functionality. Therefore, iJava implements bounds checking.

Expression evaluation order: A common source of difficulty in program porting efforts is the issue of expression evaluation order. Subexpressions that cause side effects can effect the evaluation of other parts of the same expression. If the order of evaluation of expressions is undefined, programs containing expressions with side effects cannot be reliably ported to other language implementations. Python evaluates expressions left-to-right, as does Java and JavaScript. However, C++ and Object Pascal do not define the order of expression evaluation. We leave the order of expression evaluation unspecified in iJava, a departure from Java semantics.

Expressions with side-effects: Java, C++, and JavaScript include several operators, such as the assignment operator, which can be used to modify the values of variables used in expressions. Python and Object Pascal do not include such capabilities, so iJava does not allow the use of such operations in expressions.

Table 2 summarizes the expressions and operations discussed in this section.

5.3.3 Statements

In this section, we discuss the statements found in iJava.

Assignment: Like all of our surveyed languages, iJava includes an assignment operator. Java's assignment compatibility rules are the most restrictive of our languages, and we therefore adopt them for iJava. Further, Java requires that local variables be explicitly initialized before first use, so iJava enforces this requirement.

All languages except Object Pascal support a form of the assignment operator called augmented assignment, which combines a binary operator with assignment (ex. a += b). Although Object Pascal does not support augmented assignment, since

	C++	Java	Object Pascal	Python	JavaScript	iJava
Operators						
Arithmetic	$+ - \times \div$ mod	$+ - \times \div$ mod	$+ - \times \div$ mod	$+ - \times \div$ mod x^y	$+ - \times \div$ mod	$+ - \times \div$ mod
Logical	$\wedge \vee \neg$	$\wedge \vee \neg$	$\wedge \vee \neg$	$\wedge \vee \neg \oplus$	$\wedge \vee \neg$	$\wedge \vee \neg$
Bitwise Logical	$\neg \wedge \vee \oplus$	same	same	same	same	same
Bit shift	logical, arithmetic	logical, arithmetic	logical	arithmetic	logical, arithmetic	logical, arithmetic
Relational	$= \neq > \geq$ $< \leq$	same	same	same	same	same
Evaluation Order	Undefined	Left-Right	Undefined	Left-Right	Left-Right	Undefined
Array Operations						
Length check	Y	Y	Y	Y	Y	Y
Bounds check	Y	Y	Y	Y	Y	Y
Pointer Operations						
Dereference	Y	Y	Y	Y	Y	Y
Compare	Y	Y	Y	Y	Y	Y
Address-of	Y	N	Y	N	N	N
Arithmetic	Y	N	Y	N	N	N

Table 2 – Operator and Expression Analysis

	C++	Java	Object Pascal	Python	JavaScript	iJava
Augmented Assignment	Y	Y	N	Y	Y	Y
Branch Statements						
if	Y	Y	Y	Y	Y	Y
switch	Y	Y	N	Y	N	N
break/continue	basic	labeled	basic	basic	labeled	basic
goto	Y	N	Y	N	N	N
Loop Statements						
pre-test (while)	Y	Y	Y	Y	Y	Y
post-test	Y	Y	Y	N	Y	N
for (counting)	N	N	Y	N	N	N
for (C style)	Y	Y	N	N	Y	N
for (enumerating)	Y	Y	Y	Y	partial	N

Table 3 – Statement Analysis

it can be mapped to Object Pascal in a straightforward way, we support it in iJava.

Control statements: iJava includes if/else and while constructs, the only selection and looping statements found universally in our language set. Like Java, iJava requires that if conditions and while conditions be boolean-typed.

All of our surveyed languages also include simple, unlabeled break and continue statements used to terminate loops or skip portions of loop iterations, so iJava includes these as well.

See Table 3 for a summary of the analysis in this section.

5.3.4 Subroutines

In this section, we consider subroutine-related issues including recursion, nested subroutines, scoping, parameter passing modes, and method overloading.

Recursion: iJava supports recursion, as do all of our languages.

Scoping: Object Pascal, Python, and JavaScript support nested subroutines, but C++ and Java do not (although they do support classes nested inside subroutines), so nested subroutines are excluded from our subset. Statically allocated and initialized local variables, a feature of C++, are not supported in the other languages, and are thus excluded. All of the surveyed languages utilize ALGOL-style static scoping, where every occurrence of a variable is statically associated with its definition by lexical analysis of the enclosing block structure, rather than being resolved at runtime

	C++	Java	Object Pascal	Python	JavaScript	iJava
Nested Subroutines	N	N	Y	Y	Y	N
Global Subs/Variables	Y	N	Y	Y	Y	N
Scope Resolution	static	static	static	static	static	static
Parameter Modes	value, ref	value	value, ref	value	value	value
Array Parameters	ref	ref	ref (dynamic arrays)	ref	ref	ref
Method Overloading	Y	Y	Y	N	N	Y
Recursion	Y	Y	Y	Y	Y	Y

Table 4 – Subroutine Feature Analysis

through dynamic lookup in the runtime stack.

Python, JavaScript, Object Pascal, and C++ permit globally scoped top-level variables and subroutines that are defined outside a class. Java requires all variables and subroutines to be encapsulated inside a class, so iJava does not support top-level variables and subroutines. We note that global access to variables and subroutines can be achieved in iJava through the use of public static declarations in classes.

Parameter passing modes: All of the languages support some combination of by-value and by-reference parameter passing. Java, JavaScript, and Python support only by-value parameters, but regularly achieve a form of by-reference semantics by passing references as values. iJava adopts the model employed by Java and JavaScript, in which primitives are passed by value, and values such as objects and arrays are passed by reference. This represents a reasonable intersection of the parameter passing capabilities of our language set.

Method overloading: Method overloading, a feature allowing several subroutines to use the same name, being distinguished by the number and types of their parameters, is supported in Java, Object Pascal, and C++, but not JavaScript or Python. Since method overloading can be emulated in a straightforward manner using name mangling in the latter languages, we choose to include it in iJava.

Table 4 summarizes our discussion of subroutine capabilities.

5.3.5 Object Oriented Features

All of our analyzed languages include object-oriented features, including encapsulation, inheritance, and dynamic binding. In this section, we analyze specific features in this category.

Class definition: All of our languages except JavaScript utilize a class-based approach for defining objects. In contrast, JavaScript uses a prototype approach. Since a class-based approach is inherently more restrictive than a prototype-based approach, our interoperable intersection as embodied in iJava utilizes a static, class-based approach.

Class initialization: All of our languages provide a constructor mechanism to define a block of code that executes when a class is instantiated, to initialize instance variables. iJava also supports the initialization of instance variables at the point of declaration, since this feature can be emulated using constructors in languages such as Object Pascal that do not support this feature. iJava allows the initialization of static class variables at the point of declaration, since all languages include features that can support this.

Inheritance: All of our languages support an inheritance mechanism that supports method overriding and dynamic binding. Java, JavaScript, and Object Pascal permit a class to have at most one immediate ancestor (single inheritance), while Python and C++ allow a class to have multiple immediate ancestors (multiple inheritance). Because single inheritance is more restrictive than multiple inheritance, iJava is restricted to single inheritance.

C++ and Object Pascal provide a way to define both virtual and non-virtual methods; virtual methods support overriding and dynamic binding, while non-virtual methods do not. Java, Python, and JavaScript support only virtual methods; therefore, our iJava includes only virtual methods.

Java, JavaScript, Object Pascal, and Python all utilize a unified class hierarchy, in which there exists an ultimate ancestor base class from which all other classes ultimately derive. C++ does not require a single base class, thus allowing multiple class hierarchies. In our intersection, we restrict iJava to a single class hierarchy with a common ancestor.

All of our surveyed statically typed languages have operators to perform checked downcasts. C++ and Object Pascal also provide operators to perform unsafe unchecked downcasts, but Java does not, so we exclude an unchecked downcast operation from iJava. Dynamically typed languages, by definition, do not require a downcast operator, but its behavior can be supplied via a library function.

C++ and Java both support covariant return types for overriding methods, but require invariant parameter types. Object Pascal, in addition to requiring invariant parameter types, also requires invariant return types. iJava therefore requires invariant return types as well as invariant parameter types for overriding methods.

Abstract Classes and Interfaces: Statically typed object-oriented languages use abstract classes and interfaces to enable modular development. Java, Object Pascal, and C++ all provide mechanisms to omit implementations for abstract methods in a class. Java and Object Pascal also provide a specialized form of abstract class called an *interface* that contains only abstract methods. Both Java and Object Pascal allow interfaces to form an inheritance hierarchy, and for classes to implement multiple interfaces, thereby obtaining some of the benefits of multiple inheritance. C++, which supports multiple inheritance, needs no special interface mechanism, since it can encode interfaces using pure abstract classes. These features are not provided (or needed) in Python or JavaScript, but can be emulated in these languages via library support, so we include them in our intersection. iJava supports both abstract classes and interfaces.

Information hiding: C++, Object Pascal, and Java provide enforced information hiding mechanisms for class members, using visibility modifier keywords. All three languages support the public, protected, and private keywords, with the usual semantics. Java and Object Pascal allow classes within the same namespace to freely access each others' members, in effect bypassing the visibility modifiers; in a similar vein, C++ provides a friend mechanism to bypass the access controls. For a clean, simple intersection, we adopt the C++ visibility modifier keywords public, protected, and private, and their C++ semantics, for iJava's information hiding mechanism.

Scripting languages typically do not provide an enforced information hiding feature: JavaScript lacks it entirely, and Python provides naming conventions for information hiding which can be circumvented. Since information hiding is an important language feature that can be viewed as enforcing a restriction on the set of legal programs, we consider it to be in the intersection of our language set, and support information hiding in iJava.

Table 5 summarizes our discussion of object-oriented features.

	C++	Java	Object Pascal	Python	JavaScript	iJava
Class vs Proto	Class	Class	Class	Class	Proto	Class
Inheritance	Multiple	Single	Single	Multiple	Single	Single
Interfaces	N	Y	Y	N/A	N/A	Y
Info Hiding	Y	Y	Y	Partial	N	Y
Class Hierarchy	Multiple	Single	Single	Single	Single	Single
Downcast	Safe, Unsafe	Safe	Safe, Unsafe	N/A	N/A	Safe
Methods	Virtual, Nonvirtual	Virtual	Virtual, Nonvirtual	Virtual	Virtual	Virtual
Abstract Methods	Y	Y	Y	N/A	N/A	Y

Table 5 – Object Oriented Feature Analysis

5.3.6 Exception Processing

All of our surveyed languages provide an exception mechanism to report and handle errors. In this section, we discuss the exception processing features of the languages in our set.

All of the languages use the termination model, rather than the resumption model, for exception processing: when an exception is raised, further processing in the method is terminated and a search is made for an exception handler.

Java classifies exceptions into two categories: checked exceptions thrown by a method must be explicitly handled by the caller, either by a try-catch statement or by an advertised exception propagation using a throws clause in the method interface; unchecked exceptions do not need to be explicitly handled. The other languages do not include checked exceptions, and iJava therefore includes only unchecked exceptions.

Java and Python allow only instances of a class in an Exception class hierarchy to be thrown as exceptions; C++ and JavaScript allow values of any type to be thrown; Object Pascal allows any object type to be thrown. iJava thus restricts throwable values to instances of a class in an Exception class hierarchy.

C++ and Java allow a subroutine to advertise the types of exceptions it may throw via an exception specification clause (although this feature is deprecated in recent versions of the C++ standard); the other languages do not. We therefore exclude this feature from iJava.

All of our analyzed languages include a statement to raise an exception (ex. throw in Java), as well as a statement to handle exceptions (ex. try-catch in Java). JavaScript allows only one catch block in its try-catch, while the other languages allow multiple catch blocks to provide exception-specific handling. Since multiple catch blocks are useful and can be easily emulated in JavaScript with a sequence of if-else tests that check the type of exception that was thrown, iJava allows multiple catch blocks. All of the languages that support multiple catch blocks have identical semantics regarding the resolution of matches for a given exception: the first matching catch block applies. Finally, all languages allow an exception parameter in a catch block to receive the specific exception instance that was thrown, so that it can be inspected.

All of our languages except C++ allow the try-catch construct to contain a finalization block, which executes whether or not an exception is raised. Because C++ omits this feature, we exclude it from iJava.

Table 6 summarizes the issues surrounding exceptions.

5.3.7 Other Language Features

In this section, we discuss additional language features that we considered in our analysis.

	C++	Java	Object Pascal	Python	JavaScript	iJava
Checked	N	Y	N	N	N	N
Throwable	Anything	Exception only	Any Object	Exception only	Anything	Exception only
Exception Specs	Y (deprecated)	Y	N	N	N	N
Multiple Catch	Y	Y	Y	Y	N	Y
Finally	N	Y	Y	Y	Y	N

Table 6 – Exception Feature Analysis

	C++	Java	Object Pascal	Python	JavaScript	iJava
Generics	Y	Y	Classes only	N/A	N/A	Classes only
Parameter Constraints	N	Y	N	N/A	N/A	N
Exception Specs	Y (deprecated)	Y	N	N	N	N
Instance Of Operator	Y	Y	Y	Y	Y	Y
Namespaces	Y	Y	Y	Y	N	Y
Operator Overload	Y	N	Y	Y	N	N
Threads	Y	Y	Y	Y	N	N

Table 7 – Miscellaneous Feature Analysis

Generics: The languages in our set take a variety of approaches to generic algorithms and data structures. C++, Java, and Object Pascal provide a template facility designed to facilitate generic programming, although the implementation details and capabilities differ considerably. Python and JavaScript provide generic functionality by definition, via their status as dynamically typed languages.

Object Pascal, Java, and C++ all allow classes to be defined using one or more generic type parameters, which can be used to specify types for both member variables as well as parameter and return types for member operations. Java allows constraints on type parameters to be specified, but Object Pascal and C++ do not provide a constraint mechanism. Java and C++ allow subroutines to define generic type parameters, but Object Pascal does not. Therefore, iJava allows generic classes only, with no constraints on generic parameters.

Namespaces: Many languages have structural features to prevent naming conflicts and to group code into hierarchically structured logical units. All of the surveyed languages except JavaScript provide a namespace capability. Since namespaces are regularly implemented in JavaScript programs using a variety of closure-based modular coding patterns, we support namespaces in iJava.

Introspection: The dynamic inspection of data structures is supported to varying degrees in all of our surveyed languages. All of the languages provide a mechanism to query the type of an object to determine whether it is a subclass of a specified type; examples of this operator include dynamic_cast in C++, instanceof in Java, and isinstance() in Python. Therefore, we include that operation in our subset. Other introspection features, such as dynamically discovering the fields and methods available in a given object, are not available in C++ and Object Pascal, so we exclude such functionality from iJava.

Operator overloading: Operator overloading is supported in Object Pascal, Python and C++, but not in Java or JavaScript. It is thus excluded from iJava.

Threads: All of the analyzed languages except JavaScript provide capabilities for starting and manipulating independent threads. Because JavaScript lacks support, we exclude threads from iJava.

Table 7 summarizes the discussion in this section.

5.3.8 Standard Library

We have purposefully kept the iJava standard library small in scope, to avoid limiting the potential set of compatible target languages. It includes a minimal subset of classes from the Java standard library that are needed for basic language support. It includes partial implementations of the Java wrapper classes Boolean, Character, Byte, Short, Integer, Long, and Double, which are needed to wrap primitives into objects. A subset of the java.lang.String class contains the following methods: equals, compareTo, length, substring, indexOf, and charAt, as well as the overloads of valueOf that convert from the supported primitive types to String. Finally, it includes the RuntimeException class used to report exceptions.

6 The Translation System

In this section, we discuss details concerning our translation system introduced in Section 4. Our translation system targets two of the languages in our analyzed set: Python and C++. We believe that these two languages present a reasonably representative subset of our language set, in terms of syntax, semantics, and language execution model. Targeting additional languages is part of our future work. We begin in Section 6.1 with a discussion of the mapping of Java features to iJava. Then, Section 6.2 presents the translation of iJava features to Python and C++.

6.1 Mapping Java to iJava

As we discuss in Section 7, we evaluate our system using Java test cases. These test cases use Java features which are excluded from iJava as the result of design choices imposed by our methodology described in Section 5.1. Our translation system transforms these constructs to iJava features before translating them to Python or C++. In this section, we discuss the mapping of these Java features to iJava structures. It is not our purpose to establish a complete mapping from Java to iJava; since iJava, by design, omits key primitives such as synchronization and support for multithreading, we observe that such a mapping is not possible.

6.1.1 Pre/Post Increment Expressions

iJava excludes expressions with side effects, for example the pre- and post- increment and decrement operators (ex. `++x`, `-x`, `x++`, `x-`). In cases where they occur in a statement by themselves, our translator transforms these operators to augmented assignment operations (ex. `x++` is transformed to `x += 1`). Our test cases did not make use of these operators in compound expressions.

6.1.2 Loops

iJava provides a while loop, but not Java's for, do-while, or enumerating for loop. Since our test cases made use of these loops, we transformed each of these into while loops. For example, consider a do-while loop having the structure shown in Listing 1. The translator transforms it to a iJava while loop, adding a locally defined flag variable unique to the loop (named here firstTime) that ensures that the loop body will execute at least once. iJava's short-circuiting rules ensure that the loop condition is not evaluated upon initial entry into the loop, but rather is evaluated at the end of each loop iteration that terminates normally. This transformation thus preserves the semantics of Java's do-while loop.

```
do {
  loop-body
} while ( condition );
```

Listing 1 – Java Do-while example

```
boolean firstTime = true;
while (firstTime || condition) {
  firstTime = false;
  loop-body
}
```

Listing 2 – iJava translation

6.1.3 Nested and Anonymous Classes

Our test cases contained nested and anonymous classes. Our translator converted these to standalone classes in iJava, handling the scoping issues that occur when the code in the inner class referenced members in the outer class, as is shown in Listing 3, where *innermethod()* accesses c, a member of Outer. Listing 4 demonstrates the transformation, in which the Inner class becomes a top-level class, Outer_Inner, that holds a reference to Outer, outer. This reference enables the refactored innermethod() to access c in Outer.

```
class Outer {

  int c;

  public void useInner() {
    Inner cl =
      new Inner();
    cl.innermethod();
  }

  class Inner {
    public void innermethod() {
      c = 5;
    }
  }

}
```

Listing 3 – Java inner class example

```
class Outer {

  int c;

  public void useInner() {
     Outer_Inner cl =
       new Outer_Inner(this);
     cl.innermethod();
  }
}

class Outer_Inner {

  Outer outer;

  public Inner(Outer outer) {
    this.outer = outer;
  }

  public void innermethod() {
    this.outer.c = 5;
  }
}
```

Listing 4 – iJava translation

6.2 Mapping iJava to Python and C++

In this section, we discuss the mapping of iJava features to Python and C++.

iJava	C++	Python
byte	int8_t	int
short	int16_t	int
int	int32_t	int
long	int64_t	int
double	double	float
char	char16_t	str
boolean	bool	bool
String	java_lang_String*	jcl_String

Figure 4 – Mapping of iJava types to C++ and Python

6.2.1 Classes and Interfaces

The translator maps each iJava class and interface to a class in the target language. Each member method and instance variable is mapped to a corresponding method and variable in the target class. This mapping is straightforward for C++; for Python, which does not support method overloading, name mangling is used to implement the method overloading. Additional work is required to support the mapping of overloaded constructors to Python: since a Python class may contain only a single constructor, we map iJava constructors to static factory methods in the target Python class. For an example of this mapping to static factory methods, see Line 9 of Listing 7, which shows the instantiation of an exception in a fragment of the Apache Math Commons library, and its translation on Line 12 of Listing 8, which invokes the factory method constructor_R to instantiate the translated exception class.

Information hiding: Our translator does not implement information hiding. However, we observe that, given iJava's information hiding semantics, the mapping of the visibility modifiers from iJava to C++ is straightforward. A mapping to Python would involve using Python's naming conventions for protected and private visibility—prefixing member variable names with one or two underscores, respectively.

Interfaces: Neither Python nor C++ have an interface mechanism, so we emulate the feature in both languages. For C++, iJava interfaces map to classes containing only pure virtual methods. Interface implementation is modeled using virtual inheritance. For Python, we utilize the Python Abstract Base Class module, generating classes containing methods defined with the `@abstractmethod` decorator.

6.2.2 Data Types and Variables

Figure 4 shows how iJava's basic types map to C++ and Python. We use a custom support library class to represent strings in both C++ and Python. In both cases, this class implements the Java string methods used in our test cases.

iJava arrays are mapped to C++ using a custom support library wrapper class, which handles the bounds checking. For Python, iJava arrays are modeled as standard Python lists. As an example of the way our translator handles arrays in the mapping to Python, see Listing 7, which presents a fragment of the Apache Commons Math library, and its translation to Python in Listing 8. Note how the length check on Lines 2 and 3 of Listing 7 maps to Python's `len()` function on Lines 3 and 5 of Listing 8.

```
1  for( int i=0; i<observers.length; i++ )
2  {
3      IObserver observer = (IObserver)observers[i];
4      observer.notifyObserver(note);
5  }
```

Listing 5 – Java dynamic cast example

```
1  int32_t i_ = 0;
2  while ((i_ < (observers_)->length()))
3    {
4      org_puremvc_java_interfaces_IObserver* observer_ =
5          jcl_dynamicCast<org_puremvc_java_interfaces_IObserver*>(
6              observers_->AT(i_));
7      (observer_)->notifyObserver(note_);
8      i_ += 1;
9    }
```

Listing 6 – C++ cast translation

6.2.3 Expressions and Statements

Expressions in iJava map in a straightforward fashion to C++ and Python. Some operators are implemented using library functions, in order to achieve semantic parity. For example, Line 3 of Listing 5 shows a dynamic cast that occurs in a fragment of the PureMVC library. In the translation to C++ shown in Listing 6, the dynamic cast has been mapped to an invocation of the jcl_dynamicCast library function in Line 5. All iJava statements map cleanly to their counterparts in C++ and Python.

6.2.4 Memory Management

The issue of garbage management can be addressed for C++ through the use of a garbage collection library, such as the well-known Boehm-Demers-Weiser library [BW88]. We chose not to implement it in our system; this is an area for future work.

6.2.5 Generics

iJava generics are implemented in C++ using templates, with a straightforward mapping. For Python, which needs no generic features due to its dynamic type system, generic parameters are simply removed in translation.

6.2.6 Exception Handling

iJava's exception handling constructs map in a straightforward way to Python and C++. In some cases, it is necessary to map a Java exception type to a corresponding type in the target language standard library. For example, Listing 7 shows a fragment of code from the Apache Math Commons library that handles a possible out of bounds array access in the catch block on Line 8. In the translation to Python, depicted in Listing 8, the translator mapped the ArrayIndexOutOfBoundsException type to Python's IndexError type on Line 11.

7 Evaluation

In this section, we present an evaluation of our system using two open source Java libraries as test cases for our system. In Section 7.1 we introduce the two libraries, or test cases, and in Section 7.2, we describe our approach to evaluating the test cases and preparing them for translation into Python and C++. In Section 7.3 we describe our use of the expected output of the test cases together with *execution traces* to evaluate the reliability of the generated Python and C++ code, and in Section 7.4 we describe our use of software metrics to evaluate the quality of the generated code. Finally, we conclude this section with some results that describe the performance of the generated code with respect to execution speed.

7.1 Test Cases

In this section, we introduce the two libraries that we used as test cases to evaluate our translation system: PureMVC, a Model-View-Controller (MVC) framework, and Apache Commons Math, a collection of mathematical algorithms. Both libraries include an extensive suite of unit tests implemented in the JUnit unit testing framework.

7.1.1 PureMVC

The PureMVC framework facilitates writing applications based upon the Model-View-Controller architectural pattern [Pur16]. It implements several design patterns defined by Gamma et al. [GHJV95], including Facade, Command, Mediator, Observer, and Proxy. Versions of the framework are available for multiple languages, including Java, C++, and Python. Our tool translated the entirety of version 1.1 of the Java implementation to Python and C++. In addition to its focus on design patterns, PureMVC also makes liberal use of template features.

7.1.2 Apache Commons Math

Apache Commons Math is a Java library of mathematical and statistical algorithms [Apa16]. This test case makes greater use of the Java standard library than PureMVC, including APIs such as file I/O, serialization, reflection, cryptography, and text formatting. We implemented many of these APIs in our Python and C++ support libraries to enable the conversion and testing of the majority of version 1.0 of the library in our tests.

```java
try {
    for (int i = 0; i < selectedRows.length; i++) {
        for (int j = 0; j < selectedColumns.length; j++) {
            subMatrixData[i][j] = data[selectedRows[i]][selectedColumns[j]];
        }
    }
}
catch (ArrayIndexOutOfBoundsException e) {
    throw new MatrixIndexException("matrix dimension mismatch");
}
```

Listing 7 – Java try-catch example

```
try:
  i = 0
  while (i < len(selectedRows)):
    j = 0
    while (j < len(selectedColumns)):
      subMatrixData[i][j] = self.data[selectedRows[i]][selectedColumns[j]]
      j += 1

    i += 1

except IndexError as e:
  raise org_apache_commons_math_linear_MatrixIndexException.constructor_R(
     jcl_String("matrix␣dimension␣mismatch"), None)
```

Listing 8 – Python try-except translation

	PureMVC	Apache Commons Math
Lines of Code	1,456	15,566
Modules	51	195
Test Points	100	3,027

Table 8 – Test Cases

7.1.3 Test Case Metrics

Table 8 presents metrics about the test cases. *Lines of Code* is defined as nonblank, noncomment lines of code, and includes both library code as well as the accompanying unit test code. The *Modules* count refers to individual Java source files. We use the term *Test Point* to refer to a single assertion in a unit test.

7.2 Test Case Evaluation and Preparation

In this section, we introduce the procedure that we used to evaluate and prepare the target applications that we translated using our system. This evaluation included the identification of third party libraries used by the two test cases. These third party library dependencies included Application Programmer Interfaces (APIs) supplied by the Java Standard Library and the JUnit Testing Framework. To facilitate our translation, we implemented some of these third party libraries and included the translations in our Python and C++ runtime support systems.

After implementing the required dependencies, we made minor modifications to the test cases, detailed in the next section. Finally, we used our translation system to translate the modified test case code to C++ and Python.

7.2.1 Modifications to PureMVC

We made a single modification to the PureMVC code to facilitate the translation to C++, which involved marking a class as *implementing an interface.* This omission in the original test case was an oversight in the PureMVC code, and the test case has been updated as a patch to the PureMVC library.

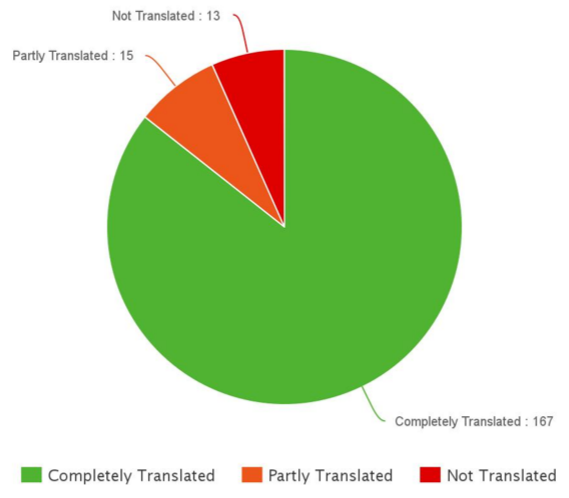

Figure 5 – Apache Commons Math Classes Translated

7.2.2 Modifications to Aapache Commons Math

In this section, we describe the modifications to the Apache Commons Math test case to facilitate our tests. Figure 5 lists the number of modules in Apache Commons Math that our tool translated in whole or in part. Our tool translated 89% of the lines of code, representing a complete or partial translation of 93% of the modules.

The 13 modules that we did not translate relied on Java APIs that we did not port to the Python and C++ runtime systems, including I/O, formatting, and reflection features. An automated translation of these APIs is part of our future work. We removed code from 15 other modules for three reasons:

1. Some code utilized Java APIs that we did not port to the Python and C++ runtime libraries, including facilities such as serialization, cryptography, and I/O.

2. A portion of the code defined two overloaded methods: one performed a computation on a float-type parameter, and the other performed an identical computation on a double-type parameter. iJava supports doubles but not floats, and our translator maps the float type to double in its translation process. This resulted in a duplicate method conflict in the translated module. We removed the float overload as superfluous in this case. If the overloads had performed different processing, and removal was not an appropriate action, a more sophisticated overload resolution strategy could have been implemented with additional effort.

3. Finally, some of the Apache Commons test code attempts to catch Null pointer exceptions. Since the behavior of null pointer dereferences is undefined in iJava, we removed those checks. An alternative approach, requiring additional engineering effort, would involve suppressing these checks automatically during translation, and this effort is part of our ongoing work.

7.3 Reliability Tests and Execution Traces

In this section, we describe the approach we used to evaluate the reliability of the Python and C++ applications automatically generated by our translation system.

The two test cases, PureMVC and Apache Commons Math, both include extensive unit tests written using the widely known JUnit Framework [JUn16]. Our basic approach to evaluating the reliability of our translation is to convert both the libraries and their accompanying unit tests to the target languages, and then to execute the translated unit tests and compare the results. A key aspect of our analysis involves collecting a *test trace* of the execution of each unit test, which consists of a log of the assertions performed by the unit test as it executes.

Section 7.3.1 describes the procedure we use to capture a trace of execution of the unit tests of the source Java test cases. Section 7.3.2 describes our methodology for executing the translated tests and capturing a test trace. Finally, Section 7.3.3 describes the approach we used to compare and evaluate the test traces.

7.3.1 Execution of Original Unit Tests

To provide a baseline for evaluating the reliability of our translator, we execute the original unit tests that are included in both of our test cases. In addition to noting the individual pass/fail results, we capture a trace of the checks performed during the execution of the unit tests. This trace contains the arguments passed to each JUnit assertion check, generally containing both expected and actual values, and shows the sequence in which the unit tests are executed by the JUnit test framework. Since JUnit has no built-in facility for capturing a trace of its unit tests, we developed a custom tool using BTrace [btr16], a JVM instrumentation facility, to capture the trace data.

7.3.2 Execution of Translated Unit Tests

After translating the test cases to the target languages, we execute the translated unit tests in order to exercise the translated library code. During the execution of the unit tests, our implementation of the JUnit Testing Framework in the Python and C++ runtime support libraries records a trace of the unit test assertions.

7.3.3 Comparison of Test Traces

After generating the three sets of test traces for both the original library and its translation to C++ and Python, we compare the test traces and analyze any differences. To compare a test trace from a translated unit test with a test trace from its corresponding original unit test, we developed a tool that filters out differences due to the way our test traces are captured and recorded in JUnit and in the test frameworks in C++ and Python.

In our comparison of the test traces, we compare each test point in the original trace with the corresponding test point in the target trace. This allows us to detect scenarios where the test framework indicates that a given test passed, but the test point values indicate that there was a difference in the expected and/or actual values for one or more assertions within the test.

As an illustration of the data captured in the test traces, Listings 9 and 10 present a portion of the trace from the Apache Commons Math original unit test and the translated Python unit test, respectively. The traces show the execution of two unit test methods: testSinZero in BisectionSolverTest, and testQuinticZero in BrentSolverTest.

```
1 ...
2 org.apache.commons.math.analysis.BisectionSolverTest.testSinZero
3 assertEquals:null‘3.141592502593994E+00‘3.141592653589793E+00‘1.000000000000000E−06
4 assertEquals:null‘3.141592621803284E+00‘3.141592653589793E+00‘1.000000000000000E−06
5 org.apache.commons.math.analysis.BrentSolverTest.testQuinticZero
6 assertEquals:null‘2.775557561562891E−17‘0.000000000000000E+00‘1.000000000000000E−06
7 assertTrue:null‘true
8 assertEquals:null‘−7.811562634330656E−11‘0.000000000000000E+00‘1.000000000000000E−06
9 ...
```

Listing 9 – Trace of JUnit Test

```
1 ...
2 org.apache.commons.math.analysis.BisectionSolverTest.testSinZero
3 assertEquals:null‘3.141592502593994E+00‘3.141592653589793E+00‘1.000000000000000E−06
4 assertEquals:null‘3.141592621803284E+00‘3.141592653589793E+00‘1.000000000000000E−06
5 org.apache.commons.math.analysis.BrentSolverTest.testQuinticZero
6 assertEquals:null‘2.775557561562891E−17‘0‘1.000000000000000E−06
7 assertTrue:null‘true
8 assertEquals:null‘−7.811562634330656E−11‘0‘1.000000000000000E−06
9 ...
```

Listing 10 – Trace of Unit Test (Python Translation)

The lines that begin with assertEquals show the arguments passed to a given invocation of the JUnit assertEquals method during the unit test, containing the expected value, the actual value computed by the function under test, and the error tolerance allowed for the comparison. The test comparison tool reports a complete match for this fragment, even though lines 6 and 8 have minor differences between the two traces due to the way the number 0 is recorded in the trace file in our JUnit trace capture and the way it is recorded in our Python trace capture file.

7.3.4 Reliability Test Results

In Table 9, we summarize the results of our reliability tests. The Java entry shows that the original unit tests included with the source Java code bases all passed. The C++ and Python results show what percentage of the test traces captured from executing the translated C++ and Python unit tests matched the corresponding test points in the test traces captured from executing the source Java unit tests.

In our PureMVC tests, our analysis of the Python traces showed that all test points matched. For C++, all of the tests that used language features defined in iJava matched. Two of the test points were generated by code that used a language feature not defined in iJava, and as expected, since our C++ compiler handled this feature differently than Java and Python, those did not match.

In our Apache Commons Math tests, our analysis of the test point value differences revealed a variety of causes for failure to match test values in the source Java unit test traces. We noted differences due to the following factors:

- **Undefined behavior:** The translated code invoked behaviors that are undefined in iJava. Examples of this include numeric overflow handling, numeric formatting, parameter evaluation order, and circular dependencies between modules.

	PureMVC Tests Passed	Apache Commons Math Tests Passed
Java	100%	100%
C++	98%	99%
Python	100%	96%

Table 9 – Reliability Test Results

- **Standard library facilities:** The implementation of our standard library facilities for Python differed from or omitted features present in the Java standard library.
- **Random number tests**: Some unit tests had behavior which was determined by the generation of random numbers, and thus produced different traces each time the unit tests were run. This meant that portions of the traces from two separate executions of the original Java unit tests did not match each other, and it follows that the corresponding test points in the Python and C++ unit test traces did not match those from the original Java tests. However, these translated unit tests passed their internal assertions, and manual inspection of the traces revealed their behavior was as expected.

7.4 Quality Tests

In this section, we discuss the approach we use to validate the quality of the translation generated by our implementation.

7.4.1 Quality Metrics

To evaluate the size and complexity of the translated code, we use two well-established measures: Lines of Code and McCabe Cyclomatic Complexity [McC76]. In order to obtain these measures using a consistent definition, we compute the metrics for the source test cases and the Python and C++ translations using *metrics* [pyt16], a tool that computes metrics for several languages. We validate the computation of the Cyclomatic Complexity metric using a tool we developed [SM14] using libclang, an interface to the Clang compiler API.

To demonstrate that the generated code preserves the class structure of the original, we compute the number of classes using information produced by Doxygen [Dox16], a documentation tool which supports our source and target languages.

7.4.2 Quality Test Results

Table 10 and Table 11 present the results of our quality metrics analysis for PureMVC and Apache Math Commons, respectively. Here, we discuss those results.

In both cases, the number of classes increased during the translation to C++ and Python. The difference is due to the handling of anonymous inner classes during translation. The metrics tool did not detect anonymous inner classes in the Java source, and those were not included in the count. During translation, the anonymous inner classes were converted to regular classes in C++ and Python, and were subsequently detected by the metrics tool and included in the count.

	Classes / Interfaces	McCabe	LOC
Java	52	37	1,456
C++	54	37	2,477
Python	54	83	1,650

Table 10 – Quality Test Results: PureMVC

	Classes / Interfaces	McCabe	LOC
Java	185	904	13,892
C++	193	904	21,397
Python	193	1,111	13,732

Table 11 – Quality Test Results: Apache Math Commons

The increase in McCabe complexity for Python in both cases is due to the translation of overloaded constructors for Python. A selection statement increased the complexity metric by 1 for each constructor, which accounts for the increase in the metric. Our Clang-based analysis tool reproduced the results produced by the Python tool.

The LOC metric for Python is very close to the Java count, but a word about the increase in the C++ LOC is in order. C++ programs split the definition of a class between .h and .cpp files, and the syntactic duplication introduced by this organization tends to increase the LOC counts for C++ programs, as compared to languages such as Java in which a class definition is contained in a single file. This duplication accounts for the larger LOC counts we observed.

7.5 Performance Tests

In this section, we discuss the performance tests we conducted on the code generated by our translator.

7.6 Performance Test Methodology

The execution performance of a program on a given language implementation is influenced by a number of factors, including the quality of the runtime library, the overhead imposed by the language's memory management, and most importantly, where the language execution model falls on the spectrum between dynamically interpreted and statically compiled systems. Therefore, evaluating the quality of a program translation from one language to another by the use of performance metrics has limited value. For example, our three languages include dynamically interpreted Python, statically compiled (C++), and Java, a hybrid; a carefully hand-tuned implementation of a given algorithm in each of these languages could be expected to have widely varying performance characteristics, due to the factors discussed above. Thus, we include performance test results here for general informational purposes, rather than for use in evaluating translation quality.

	Java	C++	Python
PureMVC	0.8	0.2	1.7
Apache Commons Math			
Linear	2.7	4.2	20.0
Analysis & Complex	3.1	0.6	5.4
Distribution	7.2	10.8	223.0
Statistics	22.9	43.5	475.0
Misc	3.6	0.5	6.2
Total Commons Math	39.5	59.6	729.6

Table 12 – Performance Test Results. All times are in seconds.

We evaluate the performance of our translations by timing the execution of the translated unit tests. Our platform is a Dell Latitude laptop with a 1.7 GHz Intel i3 processor and 8 MB RAM, running Windows 10 Professional. For the cplusplus tests, we use the Visual Studio 2013 compiler with Release configuration settings. For the Python tests, we use the standard CPython interpreter, version 3.4.2. As a baseline for comparison, we time the original Java unit tests running on the Oracle Java Virtual Machine, version 1.8.0.

In all cases, our timings exclude the initial test program load time, and include only the time to execute the unit tests. We do not take steps to exclude the warmup of the Java JIT. We measure wall clock time and average several timings to arrive at each result, excluding outlier timings from the average.

7.6.1 Performance Test Results

In this section, we present results of our performance tests.

Table 12 shows timings for the PureMVC and Apache Math Commons test cases. The numbers represent the time required to execute 500 iterations of the unit tests, in seconds. For the Math Commons test case, the times are broken down by module, for further granularity.

The PureMVC results are unsurprising. This is a relatively simple library; its operations involve basic manipulations of lists and maps. The unit tests themselves do not involve large amounts of data, and in such a scenario, we would expect the timings to favor C++, a statically compiled language. The numbers bear that out.

The Apache Math Commons results have Java the clear winner over C++ and Python, although the breakdown shows that C++ wins significantly over Java in two of the modules. The Statistics module makes heavy use of the Java TreeMap data structure, which is not available in Python or C++. Our implementation of TreeMap in C++ and Python is a list structure with linear lookup performance characteristics, and our profiling analysis indicates that the relatively poor performance numbers for that module for C++ and Python can be attributed to that data structure implementation.

8 Summary and Future Work

In this paper, we presented an interoperable object-oriented language for defining reusable software components, together with a source-to-source translation system for converting components written in this language to Python and C++. Our goal was to demonstrate that a language designed through an intersection-based analysis of a set of popular object-oriented languages is rich enough to express useful components. As our evaluation and results demonstrate, our system is capable of translating substantial test cases with good reliability and efficiency.

There are several possibilities for future work, including extending our translator to support additional target languages; handling unimplemented iJava language features; translating Java runtime support library dependencies to target languages; and integrating a garbage collection library for C++. Also, although this work focused on object-oriented languages, similar work remains to be done in the functional language space.

Source-to-source translation has long been an underutilized tool in the language interoperability chest. We believe that the problems confronting comprehensive source-to-source translators have discouraged research in this area, and trust this work has demonstrated the potential usefulness of source-to-source technology in the development of reusable software libraries.

Our system is available for download at https://github.com/sschaub/ijava.

References

[AGG+80] Paul F. Albrecht, Philip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip, and Bernd Krieg-Brückner. Source-to-source translation: Ada to Pascal and Pascal to Ada. *SIGPLAN Not.*, 15(11):183–193, November 1980. URL: `http://doi.acm.org/10.1145/947783.948658`, `doi:10.1145/947783.948658`.

[AGLM14] Pablo Arrighi, Johan Girard, Miguel Lezama, and Kévin Mazet. The GOOL system: A lightweight object-oriented programming language translator. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*, ICOOOLPS '14, pages 5:1–5:7, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2633301.2633306`, `doi:10.1145/2633301.2633306`.

[Apa16] Apache commons math. `http://commons.apache.org/proper/commons-math/`, [Online; accessed 15-January-2016].

[btr16] Btrace. `kenai.com/projects/btrace`, [Online; accessed 15-January-2016].

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988. URL: `http://dx.doi.org/10.1002/spe.4380180902`, `doi:10.1002/spe.4380180902`.

[Dox16] Doxygen. `www.doxygen.org`, [Online; accessed 15-January-2016].

[ECM15] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification.* ECMA International, 5.1 edition, [Online; accessed 3-April-2015].

[FY86] Rodney Farrow and Daniel Yellin. Translating between programming languages using a canonical representation and attribute grammar inversion. Technical Report CUCS-247-86, Columbia University Computer Science Technical Reports, 1986.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJS+15] J Gosling, B Joy, G Steele, G Bracha, and A Buckley. *The Java Language Specification Java SE 8 Edition*. Oracle, 8 edition, 2015.

[HvKPT87] R. D. Huijsman, J. van Katwijk, C. Pronk, and W. J. Toetenel. Translating Algol 60 programs into Ada. *Ada Lett.*, VII(5):42–50, September 1987. URL: `http://doi.acm.org/10.1145/36077.36080`, `doi:10.1145/36077.36080`.

[Int14] International Organization for Standardization. *ISO International Standard ISO/IEC 14882:2014(E) Programming Language C++*. ISO, 2014.

[JUn16] Junit. `http://junit.org/`, [Online; accessed 15-January-2016].

[KLV03] Justin Koser, Haakon Larsen, and Jeffrey A Vaughan. SML2Java: a source to source translator. In *Draft Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages*, page 105. Citeseer, 2003.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, 2nd edition, 1999.

[McC76] T. J. McCabe. A complexity measure. *IEEE TSE*, 2(4):308–320, December 1976.

[MW91] Vincent D Moynihan and Peter JL Wallis. The design and implementation of a high-level language converter. *Software: Practice and Experience*, 21(4):391–400, 1991.

[pas15] Free Pascal: Reference guide. `http://freepascal.org/docs-html/ref/ref.html`, [Online; accessed 3-April-2015].

[Pla05] David Plaisted. An abstract programming system. In Susan Shannon, editor, *Leading-Edge Computer Science*, pages 85–129. Nova Science Publishers, 2005.

[Pla13] David A Plaisted. Source-to-source translation and software engineering. *Journal of Software Engineering and Applications*, 6:30–40, 2013.

[Pro02] J Prosise. *Programming Microsoft .NET*. Microsoft Press, Redmond, 2002.

[Pur16] Puremvc. `http://puremvc.org`, [Online; accessed 15-January-2016].

[pyt15] The Python language reference. `https://docs.python.org/3.4/reference/index.html`, [Online; accessed 3-April-2015].

[pyt16] metrics. `https://pypi.python.org/pypi/metrics`, [Online; accessed 15-January-2016].

[SM14] Stephen Schaub and Brian A Malloy. Comprehensive analysis of c++ applications using the libclang api. In *International Society of Computers and Their Applications (ISCA)*, 2014.

[Swi15] Simplified wrapper and interface generator. `http://www.swig.org/`, [Online; accessed 3-April-2015].

[TIO15] TIOBE Software: Tiobe index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, [Online; accessed 3-April-2015].

[TO98] Andrew Tolmach and Dino P Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(04):367–412, 1998.

[TOFN11] Marco Trudel, Manuel Oriol, Carlo A Furia, and Martin Nordio. Automated translation of Java source code to Eiffel. In *Objects, Models, Components, Patterns*, pages 20–35. Springer, 2011.

[TV00] Andrey A Terekhov and Chris Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.

[WDH89] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 114–122, New York, NY, USA, 1989. ACM. URL: `http://doi.acm.org/10.1145/74850.74862`, `doi:10.1145/74850.74862`.

About the authors

Stephen Schaub is an Assistant Professor of Computer Science at Bob Jones University. He is a member of the Association for Computing Machinery. Contact him at `sschaub@bju.edu`.

Brian A. Malloy is an Associate Professor of Computer Science at Clemson University. His research focus is front-end compiler technology and program analysis. His interests include the utilization of compiler technology to perform reverse engineering of a program to a graphical representation of the code. This graphical representation is then exploited to perform analysis for comprehension, program visualization, testing, or program maintenance. Contact him at `malloy@cs.clemson.edu`, or visit `http://brianmalloy.com`.