

# Strategies for Migration from C to Java

Johannes Martin and Hausi A. Müller  
University of Victoria  
Department of Computer Science  
Victoria, BC, Canada  
{jmartin,hausi}@csr.csc.uvic.ca

## Abstract

*With the growing popularity of the Java programming language for both client and server side applications in network centric computing, there is a rising need for programming libraries that can be easily integrated into Java programs. Many mature programming libraries exist for the C programming language. This paper surveys approaches to the migration of C source code to Java and evaluates their usefulness with respect to the integration of the migrated code into Java programs.*

## 1. Introduction

Electronic commerce over the Internet plays an important role in today's economy. In order to stay competitive in the global marketplace, companies have to offer their services and products to current and prospective customers online through Internet clients.

For many applications, for example browsing product catalogues or obtaining account balances from a financial institution, it is sufficient for the Internet clients to access the data stored on the company's information systems. As soon as value-added services are to be offered, it is desirable to have the Internet client not only access data from the company's information systems but also perform computations on the data. Offloading these computations from the central servers helps keeping the servers available for other tasks. If the Internet client can perform the computations independently, delays through network congestion or heavy load on the central servers can be avoided, thus improving customer satisfaction.

Internet clients commonly run as part of a web browser and are therefore written in Java [6], the programming language supported by most web browsers. To make communication between the client applications and the servers easier, companies have started to also partially or fully reimplement the server applications in Java. In order to avoid

a complete redevelopment of the business logic already present in the current server applications, it is desirable to integrate parts of these server applications – usually written in a legacy programming language such as COBOL, Fortran, or C – into the new Java clients and servers.

This paper first presents the basic technologies for reengineering legacy systems before surveying in more detail a number of strategies for integration of legacy C code into Java programs. It explains the most important differences between C and Java that make integration difficult and shows how they are addressed by these strategies. It also presents a new approach that tries to solve some of the deficiencies of the existing tools.

This migration approach focuses on programming libraries which do not depend heavily on system or runtime libraries of the source system. Based on assumptions about these libraries, we choose some conversion strategies over others. If complete applications or libraries that depend on runtime libraries are to be migrated, some of these decisions might have to be reconsidered, and additional (possibly extensive) efforts may be required to either find replacement runtime libraries or to migrate these runtime libraries to Java as well.

## 2. Differences between C and Java

The most important difference between C and Java is the hardware platform for which these programming languages are compiled. C programs are usually compiled to the native machine language of a computer, while Java programs are compiled for the *Java Virtual Machine* [12] (JVM), a virtual hardware platform running on top of a concrete hardware platform. While the basic instructions of a computer's native machine language and the machine language of the JVM are similar, the JVM also takes care of type conversions and memory management. The JVM checks all type conversions and storage accesses for their safety and security and imposes conservative restrictions on these operations. C compilers, on the other hand, implement type

conversions and storage access using the native machine language of the target computer in a very lenient but efficient way, assuming that the programmer has made sure that these operations are safe and secure.

C programmers have often taken advantage of the leniency of C compilers and their knowledge of the concrete hardware architecture of the computer for which they are writing programs, and written code that, even though it seemingly violates type safety, runs correctly and efficiently. The challenge in integrating C code with Java programs is to reconcile the type systems of C and Java and to find standard techniques to convert type lenient C code to type safe and semantically equivalent Java source.

### 3. Survey and Evaluation

Two main approaches exist for the integration of C code with Java: On the one hand, the Java Virtual Machine can be extended using code compiled to the native machine language of the target system. Section 3.1 discusses this approach. On the other hand, the C code can be compiled for the Java Virtual Machine, either directly (Section 3.2) or by first converting the C source to Java. This conversion can be done either by translating the C data types and functions to mostly equivalent Java data types and functions (*transliteration*, Section 3.4), or by recovering the design and algorithms used in the C code and reimplementing them in Java (Section 3.3).

#### 3.1. Integration of Native Binary Code

Being a virtual machine, the JVM does not execute programs at the speed that could be obtained with programs compiled to native machine language and does not allow access to many features particular to a specific concrete hardware platform.

In order to allow developers to implement performance critical code in the native machine language of a computer (either by coding it in assembly or using a native compiler) and to take advantage of features of a hardware platform not exploited by the JVM, the *Java Native Interface* [11] (JNI) has been designed.

Migration of C code to be integrated into Java programs using the JNI is easy in that no or little modifications are necessary to the C source code. However, it is necessary to provide interface classes that handle the communication between the C and Java parts of the program. While the generation of these interfaces can be largely automated, they constitute a major performance overhead at runtime, as the interfaces frequently have to perform data type conversions. One has to carefully evaluate whether the performance gain through the implementation of code in C justifies the performance loss incurred through the interfaces.

A major drawback in the use of JNI is the loss of platform independence. Since the C code has been compiled to the native machine language of a particular hardware platform, the combined C/Java program will only run on this particular platform. This may be acceptable for server applications that run on only one particular platform but is unsuitable for client applications that usually need to run on various different platforms.

Since the C code is not executed within the safe environment of the JVM, it is susceptible to failures and security breaches that could be prevented if it was running under the control of the Java Virtual Machine.

#### 3.2. C to ByteCode Compilation

Though the Java Virtual Machine was designed with the Java programming language in mind, it is general enough to support many other programming languages. A comprehensive list of programming languages available for the JVM can be found in [20]. The *Java Backend for GCC* [21] implements a C compiler that generates machine language for the Java Virtual Machine. While this appears to be a good strategy, it has been implemented in a way that makes integration with Java programs difficult and circumvents Java's type safety.

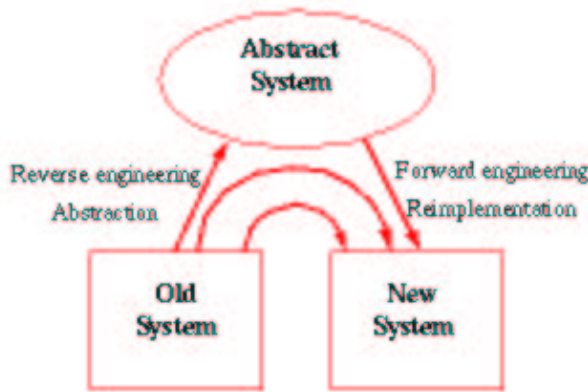
In order to accurately map the memory handling of C runtime environments onto the Java Virtual Machine and to work around the strict type checking of the JVM, a large array is allocated that is used to store all variables used in the C source code. As in a C program compiled to native machine code, faulty programs may cause corruption of this entire array, thereby causing unexpected errors that are difficult to debug. One can argue that this problem is negligible when dealing with mature programming libraries, but even these have the occasional bug and need to be enhanced and maintained.

Another problem with this approach is that special interfaces are needed in order for the Java code to communicate with the C code. Support routines are needed to build Java objects out of the raw data in the memory array. As with the JNI, these conversions create a possibly significant runtime overhead.

#### 3.3. Reimplementation

One way to turn source code from one programming language into another is to use the original design documents of the code and follow a forward engineering approach to reimplement the code using these documents in the new target language. In many cases however, these original design documents do no longer exist, or do not reflect the real architecture and functionality of the system: As Lehman states in his *Laws of Software Evolution* [9, 10], software

“systems must be continually adapted else they become less satisfactory” (*Law of Continuing Change*). Because of time pressures, design documents are rarely kept up to date with the development of the code. In this case, the current design needs to be recovered from the source code (*reverse engineering*). Once the design has been documented, it can be implemented in the target language, possibly after refinement and adaptation to features of the target language or anticipated changes. Depending on the degree of redesign desired, the abstraction can be taken to different levels (see Figure 1).



**Figure 1. Levels of Reengineering**

This approach is often taken if the existing system has maintenance or performance problems, or large changes to the requirements are anticipated. A number of papers that describe this approach have been published [2, 3, 17]. Usually only parts of this process can be automated. Opdyke [16] and Fowler et al. [5] present *refactoring* as a means for architecture refinement that can be well automated.

As we are looking for an automated approach that is suitable for converting large amounts of source code, and have no imminent desire to change the design of the source code, we did not pursue this option, but focussed our attention on source code transliteration.

### 3.4. Source Code Transliteration

With the transliteration approach, the original source code is converted to the target language, while changing the data structures and program logic as little as possible. Various degrees of change are possible: by emulating the source language’s data types in the target language (*data type emulation* [22]), the amount of change can be kept low. If data types used in the original source are substituted by data types of the target language, the code may have to be changed more in order to deal with the differences of the

```

struct s1 {
    int i;
};

struct s1 foo(int *i, struct s1 s) {
    while (s.i++, *--i > 0) {
        int h;
        for (h = *i; h < s.i; h += *i)
            if (h % 3 == 0)
                goto continue_while;
            else
                if (h % 2 == 0)
                    goto break_while;
        continue_while:
    }
    break_while:
    s.i += *i;
    return s;
}

void bar() {
    struct s1 s = { 5 };
    struct s1 t;
    int* pi = malloc(100 * sizeof(int));
    int i;
    for (i = 0; i < 100; i++)
        pi[i] = i;
    t = foo(pi + 100, s);
}

```

**Figure 2. Sample C Program**

data types. Kontogiannis et al. [7] present the results of a case study in a highly automated language conversion from PL/I to C++.

To better explain the differences between the approaches to transliterate C source code to Java, we present a small sample C program, that exhibits some of the difficulties in C to Java migration (Figure 3.4). In particular, it shows pointers to primitive data types, structure assignments, and function calls with non-primitive call-by-value parameters. It also employs goto statements and comma expressions.

#### 3.4.1. C2J++

C2J++ [8, 19] is a tool for converting C++ classes to Java classes. It transliterates data types and control flow quite well as long as they are similar in C++ and Java, but struggles where there are differences.

Figure 3 shows the transliteration result for the example C source code as produced with C2J++. As C2J++ expects all methods and variables to be within a class,

```

class s1 {
    int i;
}

class foobar {
    public s1 foo(int i, s1 s) {
        while (s.i++, --i > 0) {
            int h;
            for (h = i; h < s.i; h += i)
                if (h % 3 == 0)
                    goto continue_while;
                else
                    if (h % 2 == 0)
                        goto break_while;
            continue_while:
        }
        break_while:
        s.i += i;
        return s;
    }

    public void bar() {
        s1 s = { 5 };
        s1 t;
        int pi =
            malloc(100 * sizeof(int));
        int i;
        for (i = 0; i < 100; i++)
            pi[i] = i;
        t = foo(pi + 100, s);
    }
}

```

**Figure 3. C2J++ Transliteration Result**

the sample code was slightly modified before the transliteration. C2J++ transliterates the C source into a syntactically mostly correct Java program. It fails to recognize and convert comma expressions and goto statements. Data type conversions are done in a trivial way, usually by removing address and dereference operators. C2J++ fails to distinguish between a star as used in a multiplication and a star used to dereference a pointer, and simply removes any star it encounters. The expression inside the `malloc` statement in our example exhibits this problem. The different assignment and parameter passing semantics of C and Java are ignored, with the result that the transliterated program has a very different behaviour than the original.

C2J++ flags the changes it has made with comments (they have been removed in Figure 3 to increase readability), so the programmer can review and correct the transliterated code. While C2J++ can assist a developer in the integration of C code into Java programs, extensive manual efforts are necessary to transliterate large volumes of source code.

### 3.4.2. C2J

Novosoft's C2J [15] is similar to C2J++ in that it transliterates C source to Java, but it does not handle C++. It has been applied successfully to nontrivial programs and solves many of the problems that C2J++ struggles with. It also transliterates control flow features of C that are not supported by Java (such as comma expressions and goto statements). C2J comes with a large C runtime library providing most of the routines that typical C programs require.

Figure 4 shows a part of the transliteration generated by C2J. The transliterated source is much longer than the original. The logic of the program is difficult to understand, since many complex transformations have been applied to exactly emulate the behavior of the original C source (the source code shown has already been simplified by removing dead-code and superfluous nesting).

C2J shares some of the disadvantages of the Java Backend for GCC: Data structures are stored in a large array, thus circumventing Java's type checking and runtime security checks. The array access required for many operations and the sometimes necessary extra type conversions create a runtime overhead. They also make the code difficult to read. In this sense, C2J provides only little advantage over the Java Backend for GCC: debugging might be easier with the transliterated Java source code available, but on the other hand, some C control flow structures may be more efficiently implemented by compiling the C source code directly to machine language for the JVM.

```

class sample {
public int cfoo(int ci, int cs) {
    nextlevel();
    int label = 0; int retval = calloca(4);
    int ch_5 = 0; int y1 = 0; label = 0;
break_while:
    switch(label) {
    case 0:
        label = -1;
    lab_sample0:
        while(true) {
            sincMEMINT((int)((cs + 0)),+1);
            if (((getMEMINT((int)((ci-= 4))))>(0))?1:0)==0 )
                break lab_sample0;
            label=0;
            do {
            continue_while:
                switch (label) {
                case 0:
                    label =- 1;
                    ch_5 = (int) getMEMINT((int)(ci));
                lab_sample1:
                    for ( ; (((ch_5)<(getMEMINT((int)((cs + 0))))?1:0)!=0 ; ) {
                        if ( (((((int)((ch_5)%3)))==(0))?1:0)!=0) {
                            label = 1;
                            break continue_while;
                        } else if ( (((((int)((ch_5)%2)))==(0))?1:0)!=0) {
                            label = 2;
                            break break_while;
                        }
                    }
                    ch_5= (int)((int)((ch_5) + (getMEMINT((int)(ci)))));
                }
                case /*continue_while*/ 1:
                    label = -1;
                }
            } while (label != -1);
        }
    case /*break_while*/ 2:
        y1 = (int)((cs + 0));
        setMEMINT((int)(y1),(int)((int)((getMEMINT((int)(y1)))
            + (getMEMINT((int)(ci))))));
        retval = ((int)cs);
        prevlevel();
        return retval;
    }
}
/* ... */
}

```

**Figure 4. C2J Transliteration Result (excerpt only, hand optimized)**

## 4. Ephedra - A New Approach

The previous sections pointed out limitations and problems of the existing approaches to integrating C source code into Java programs: C2J++ requires extensive manual work in the verification and correction of the transliterated code, while JNI, C2J, and the Java Backend for GCC are susceptible to compromises to Java's type safeness and security, along with possibly large performance overheads.

To find a better solution to the problem of integrating C source code into Java programs, we are developing Ephedra [13], an environment for migration of C source code to the Java Virtual Machine. Ephedra reads C source code and produces Java source code. In order to minimize user involvement, Ephedra covers a high percentage of the C language, and directs the user where user intervention is required. The Java source code produced does not circumvent the safety features of the Java Virtual Machine and can be easily integrated with Java programs. While the emphasis is on the C language, the most commonly used language elements of C++ are also supported by the tool.

Ephedra benefits from the experiences gained in a pilot study [1], in which parts of a large C++ application were transliterated to Java.

Figures 5 and 6 show the transliteration generated by Ephedra. In the following sections, we describe some of these and other transformations done by Ephedra and show how they map behavior of the original C source to Java without compromising the JVM's integrity while at the same time facilitating the integration of the transliterated code into Java programs. As our focus is on C, we do not explain the conversion of any C++ language features.

### 4.1. Data Types

While the primitive data types of C and Java are very similar, Java does not have unsigned data types (except for the `Char` data type). Ephedra replaces all unsigned data types by their signed equivalent (or optionally next larger integral data type). The user of the tool is notified of these changes, so she can review the transformations to make sure that they don't modify the behavior of the code. In many cases this will require the developer to have a deep knowledge of the code to be converted, so she can judge whether differences in the boundary conditions in C and Java (data type overflows) will have an effect on the runtime behavior of the code. Our observation is that many C programs do not use the full range of the primitive data types and thus do not change their behavior if the type is only slightly changed. The programs that rely on the boundary conditions of specific data types need to be hand-modified.

As a means for reducing the amount of manual labor required in a migration, Terekhov and Verhoef [18] suggest

the use of data type emulation, in which the original unsigned data types would be emulated through classes in Java or by instrumenting the code operating on these data types. It has the advantage that the conversion can be fully automated with no need for a developer to understand the original code in order to validate the conversion. We rejected this option for the conversion of primitive data types in order to improve the performance and readability of the generated code: a language conversion is a long-term investment and should emphasize on maintainability.

Structures are trivially transformed into classes with public fields. Where structures contain other non-primitive data types, the resulting Java classes contain references to the equivalent Java classes. Default constructors are added to the class definitions to initialize these references.

An approach that can be applied to transforming C unions to Java has been presented in [14].

### 4.2. Variable Declarations and Initializers

For variables of primitive types, no transformation other than the conversion of the data types is necessary. The type of variables representing pointers to primitive data types is changed into corresponding wrapper types. Also, if the address of a primitive variable is taken in the C source code, a wrapper type is used in the transliteration, with an appropriate initializer allocating memory for the wrapper variables.

Structured variables are replaced by references to classes defining the data structures in Java. An initializer is added to allocate storage for the class. In C, fields of a structure can be initialized using so called *brace list initializers* that are not supported for classes in Java. Constructors that take initializers for every field of a class as parameters are used to replace these brace list initializers. Figure 6 shows the conversion of declarations and initializers in the `bar` method.

Pointers to structured data types are also replaced by references to the corresponding classes, but no transformation has to be done for the initializers.

### 4.3. Assignments and Function Calls

Again, we need to distinguish between primitive and structured data types. No transformation is necessary for assignments and function calls with variables of primitive data types. If pointers to primitive data types are assigned or passed in the C source code, references to the equivalent wrapper classes will be used in the transliteration.

In order to emulate a C language structure assignment in Java, methods similar to assignment operators in C are defined for the class involved in the assignment. The assignment is replaced by a call to the assignment method. Similarly, to pass structures using by value to a method, a constructor similar to a C++ copy constructor is defined

```

class Int {
    public int value;
    private Int[] containingArray;
    private int indexWithinArray;

    public Int() { }
    public Int(Int[] containingArray, int indexWithinArray) {
        this.containingArray = containingArray;
        this.indexWithinArray = indexWithinArray;
    }
    public Int arrayDereference(int relIndex) {
        int newIndex = relIndex + indexWithinArray;
        if ((newIndex < 0) || (newIndex >= containingArray.length))
            return new Int(containingArray, newIndex);
        return containingArray[newIndex];
    }
    public static Int[] makeArray(int size) {
        Int[] temp = new Int[size];
        for (int i = 0; i < size; i++)
            temp[i] = new Int(temp, i);
        return temp;
    }
}

```

**Figure 5. Ephedra Builtin Int Class**

for the class being passed. For pointers to data structures, or call-by-reference parameters, no transformation is necessary. Figure 6 shows the constructors and assignment methods for the `s1` class.

#### 4.4. Classification and Conversion of Pointers

By investigating C programs, one can find several common uses of pointers, that can easily be mapped to Java:

**call-by-reference parameters** In both C and Java, parameters are generally passed by value. In C, the effect of passing a variable by reference is achieved by passing the address of this variable. In Java, all non-primitive types are effectively passed by reference, since they are accessed through references. To pass variables of primitive types by reference, they need to be wrapped into classes.

**array traversals** C programmers often use pointers to elements of an array in order to traverse these arrays by incrementing the pointers. In Ephedra this behavior is emulated by including with every object contained within an array a reference to the containing array and the index of the object within that array. This approach can be used to exactly map the runtime behavior of C array traversals with the advantage that Java's bounds

checking prevents memory access violations. Figure 5 shows this technique for the integer wrapper class.

In a typical case of an array traversal in C, a pointer may point to an address outside the array – such as the first memory location just after the array. The `arrayDereference` method of `Int` creates objects to represent these addresses when needed.

The `makeArray` method builds an array of integer wrapper objects while initializing the references and indices from the array elements into the array object.

If pointers to non-primitive classes are used in the program, similar code is added to the definition of these classes. As these array traversals are quite common in C code, the data type emulation option (see 4.1) is here more feasible than a simpler transliteration that would require extensive manual verification.

**dynamic memory** Pointers in C have been used to access dynamically allocated memory. The same effect can be achieved in Java using references.

**polymorphism** As C has very limited builtin support for polymorphism as it is known from object-oriented languages, some programmers have used pointers to achieve a similar effect in their programs. A technique and algorithms to detect and transform such hidden

```

class s1 {
  public int i;
  public s1()          { }
  public s1(int i)     { this.i = i; }
  public s1(s1 _s1)    { assign(_s1); }
  public s1 assign(s1 src) { this.i = src.i; return this; }
}

class foo {
  public static s1 foo(Int i, s1 s) {
  while_001:
    while (((s.i++ | 1) != 0) && ((i = i.arrayDereference(-1)).value > 0)) {
      int h;
      for (h = i.value; h < s.i; h += i.value)
        if (h % 3 == 0)
          continue while_001;
        else
          if (h % 2 == 0)
            break while_001;
    }
    s.i += i.value;
    return s;
  }
}

class bar {
  public static void bar() {
    s1 s = new s1(5);
    s1 t = new s1();
    Int[] pi = Int.makeArray(100);
    int i;
    for (i = 0; i < 100; i++)
      pi[i].value = i;
    t.assign(foo.foo(pi[0].arrayDereference(100), new s1(s)));
  }
}

```

**Figure 6. Ephedra Transliteration Result**

polymorphisms by following type casts has been presented in [14] and implemented for Ephedra.

Some type conversions achieved through the use of pointers are not portable and cannot generally be mapped to Java:

**casts between pointers to integers and floats** These casts have in some cases been used to retrieve parts, such as the exponent, of a floating point number. As the representation of floating point numbers in memory is hardware-dependent, a tool cannot generate a generally valid transformation. Rather, a developer has to investigate the code and find a correct mapping.

**casts between pointers to data structures and `void*`**

Frequently used with calls to library functions responsible for memory management and I/O. These casts are not portable – their results depend on the specific hardware a program runs on. Ephedra recognizes certain library functions dealing with memory management, such as `malloc` and `memcpy` and replaces them by calls to equivalent Java methods. Since I/O in Java is quite different from I/O in C, the programmer will have to perform manual transformations for these cases.

By not using the approach of storing all C data structures in a large array (as C2J and the Java Backend for GCC do), Ephedra is unable to perform an automatic transformation for some of these unportable situations. This disadvantage is justified by the much better readability and type safety of the code generated by Ephedra.

#### 4.5. Labels and `goto` Statements

Java does not support a `goto` statement. Even though C supports it, it has been used very rarely, as its use often leads to poorly readable code. In some cases, `goto` statements have been used to get around limitations of the C language, and Ephedra concentrates on detecting and transforming these cases, while asking the developer to improve code that uses `goto` statements where they shouldn't be used.

We identify the following classes of `goto` uses and transformations:

**nested loops** `break` and `continue` statements in C always break or continue the innermost loop only. Breaking out of or continuing an outer loop can be achieved through `goto` statements. In Java, loops can be labeled and `break` and `continue` statements can refer to these labels to signify which loop in a set of nested loops should be broken or continued. The `foo` function in our sample program employs these strategies. Figure 6 shows their transliteration.

**`goto` to the end of a function** This scenario is often used in order to avoid coding cleanup code multiple times in a function. A label is put before the cleanup code, and instead of a `return` statement a branch to that label is used to return from the function. In Java, the same effect can be achieved by putting a label on a lexical block extending from the beginning of the function to just before the labeled statement in the C source and then using a labeled `break` statement.

Ephedra does not transform other `goto` statements that break the flow of control. As C2J proves, such a transformation is possible, but it usually results in poorly readable code. Our opinion is that a developer should rewrite such code in a proper manner.

#### 4.6. Comma Expressions

C allows for multiple expressions to be chained together into one expression using commas. The outer while loop in the `foo` function of our sample code shows this scenario. Expressions of this kind are frequently used in C macros. All subexpressions are evaluated, but only the result of the final subexpression is used. Java does not support this kind of expression (except in `for` statements), but a transformation is not difficult: all but the last subexpression are augmented to be always true, and all subexpressions are concatenated with logical AND operators.

If one of the subexpressions has a `void` value, it cannot be augmented to become a true boolean expression. In this case, the subexpression has to be wrapped into a method that always returns `true`, and a call to this method is inserted into the comma expression.

#### 5. Conclusion

In this paper, we showed why the integration of C and Java programs has gained importance. We presented various current strategies for this integration and explained their main characteristics, strengths and weaknesses. We introduced a new approach and pointed out how it addresses and solves the weaknesses of previous integration strategies.

#### References

- [1] A. Agrawal. Converting C++ programs to Java — A Case Study. Master's thesis, University of Victoria, 1999.
- [2] P. Aiken, O. K. Ngwenyama, and L. Broome. Reverse-engineering: New systems for smooth implementation. *IEEE Software*, pages 36–43, Mar. 1999.
- [3] M. I. Cagnin, R. Pentead, R. T. V. Braga, and P. C. Masiero. Reengineering using Design Patterns. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 118–127, Brisbane, Queensland, Australia, Nov. 2000.

- [4] H. Erdogmus and O. Tanir, editors. *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation*. Springer-Verlag New York, Inc., Dec. 2001.
- [5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman, Inc., 1999.
- [6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., 1996.
- [7] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. A. Müller, and J. Mylopoulos. Code Migration Through Transformations: An Experience Report. In *Proceedings of CASCON '98*, pages 1–13, Toronto, ON, 1998.
- [8] C. Laffra. C2J, a C++ to Java translator. (web site no longer available).
- [9] M. M. Lehman. On Understanding Laws, Evolution and Conversation in the Large Program Life Cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.
- [10] M. M. Lehman. Programs, Life Cycles and Laws of Software Evolution. *Proceedings of the IEEE Special Issue on Software Engineering*, 68(9):1060–1076, Sept. 1980.
- [11] S. Liang. *The Java Native Interface, Programmer's Guide and Specification*. Addison-Wesley Longman, Inc., 1999.
- [12] F. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Inc., 1999.
- [13] J. Martin. *Ephedra: A C to Java Migration Environment*. PhD thesis, University of Victoria, 2002 (in preparation). <http://www.rigi.csc.uvic.ca/~jmartin/Ephedra>.
- [14] J. Martin and H. A. Müller. *Discovering Implicit Inheritance Relations in Non Object-Oriented Code*, chapter 11. In Erdogmus and Tanir [4], Dec. 2001.
- [15] Novosoft. C2J — C to Java translator. <http://www.novosoft-us.com/NS2B.nsf/w1/C2J>, September 2001.
- [16] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. <http://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z>.
- [17] P. Patil, Y. Zou, K. Kontogiannis, and J. Mylopoulos. Migration of Procedural Systems to Netowkr Centric Platforms. In *Proceedings of CASCON 1999*, pages 68–82, Toronto, ON, Nov. 1999.
- [18] A. A. Terekhov and C. Verhoef. The Realities of Language Conversions. *IEEE Software*, pages 111–124, Nov. 2000.
- [19] I. Tilevich. Translating C++ to Java. *First German Java Developers' Conference Journal*. <http://sol.pace.edu/~tilevich/c2j.html>.
- [20] R. Tolksdorf. Languages for the Java VM. <http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html>, November 2000.
- [21] T. Waddington. Java Backend for GCC. <http://archive.csee.uq.edu.au/~csmweb/uqbt.html#gcc-jvm>, November 2000.
- [22] K. Yasumatsu and N. Doi. SPiCE: A System for Translating Smalltalk Programs into a C Environment. *IEEE Transactions on Software Engineering*, 21(11):902–912, 1995.