

C to Java Migration Experiences

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. In a previous paper, we surveyed current strategies for integrating C source code into Java programs, pointed out their weaknesses and presented goals for an improved migration approach. In this paper, we present the Ephedra approach to software migration and report on the results of three case studies transliterating C source code to Java using the Ephedra environment.

1. Introduction

Electronic commerce over the Internet plays an important role in today's economy. 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, like 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 [10], the programming language supported by most Web browsers. To make communication between the client applications and the servers eas-

ier, companies have also started to re-implement the server applications partially or fully in Java. 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.

In a previous paper [22], we investigated several strategies for migrating legacy C code to the Java platform and determined whether they were suitable for integrating the C code into Java programs. We identified some deficiencies of these strategies and presented goals and considerations for an improved migration approach called *Ephedra*. We have since developed Ephedra further and implemented a set of tools to aid a software engineer in the migration process. In this paper, we briefly summarise the main points of the previous paper, explain the foundations of Ephedra, and show the results of three case studies transliterating C source code to Java using Ephedra.

2. Related Work

Migrating source code from C to Java is a hard problem with many facets: as C is a procedural language and Java is an object-oriented language, not only the syntax and semantics of the source code need to be translated, but also a paradigm shift is necessary to move from procedural to object-oriented code. The following sections review related software migration approaches.

2.1. Language Conversion

In their paper *The Realities of Language Conversions* [30], Terekhov and Verhoef give an account of their experiences with language conversions. The examples they provide deal mostly with COBOL systems and generalise to many instances of language conversions. They argue that the difficulties of such conversions are often underestimated and manifold: Too much emphasis is put on technology and

tools that claim to aid in language conversion and too little attention is paid to training of the personnel that has a major impact on the success or failure of the migration project.

They articulate a number of important facts about source conversion, namely:

- Candidates for language conversion are usually the most critical systems of a business; thus, an emphasis must be put on the reliability of the conversion process.
- The software engineers performing the conversion must be experts in both the source and target languages to realise the intricate differences between the language and the problems that can arise out of them.
- A converted system will not be as well designed as a system developed specifically for the target programming language.
- The more similar the source and target languages are, the more difficult will be the detection of their differences: syntactically close or identical source artifacts might have big semantic differences.
- It is very difficult or even impossible to go from a rich source language to a minimal target language.

Terekhov and Verhoef list several requirements that have to be met to achieve a successful source conversion. They also propose a coarse three step process for source conversion.

In *The Migration Barbell* [19], Malton notes that there are many ad-hoc techniques for source conversion, but few systematic approaches. He formalises the process proposed by Terekhov and Verhoef. Malton also defines a set of goals for source conversion and identifies three distinct conversion tasks, which are listed in ascending order of complexity:

Dialect conversion is the conversion of a program from one dialect of a programming language to another dialect of the same programming language. This usually has to be done, if a new version of a compiler is used to build the system, or if a different compiler product is selected.

API migration is the adaptation of a program to a new set of APIs. This occurs for example, if a different database or user interface is chosen for an information system.

Language migration is the conversion from one programming language to a different one possibly involving dialect conversion and API migration.

Malton's observations are founded on dialect conversions in the COBOL, PL/I, and RPG domains, and pilot studies in

source conversion from COBOL to Java, RPG to COBOL, and SQL to SQLj.

There are a number of papers describing experiences and lessons learned from source conversion projects. Kontogiannis *et al.* [13] report on the conversion of the IBM compiler back-end from a PL/I derivative to C++ Yasumatsu [34] describes a system for translating Smalltalk programs into a C environment. Terekhov [29] presents a case study on an automated language conversion project from a proprietary language to Visual Basic and COBOL.

2.2. Paradigm Shift

Another level of complexity is added to source conversion if it involves a paradigm shift. With the growing popularity of object-oriented languages, there have been many attempts to move from procedural to object-oriented systems. One of the major problems in this particular paradigm shift is the identification of candidates for classes and their members.

A common approach is to use data structures in the legacy system as basic building blocks for classes and to add functions that operate on these structures as methods to those classes [3, 17, 18, 35, 13].

A different approach is to use design documents of a subject legacy system, such as structure charts and data-flow diagrams, to recover a possible object-oriented architecture for the subject system [8, 9].

Cimitile *et al.* centre the identification of classes around persistent data stores, such as files or tables in a database, with functions as candidate methods for these classes [4].

There are other paradigm shifts that may occur concurrently with the shift from procedural to object-oriented code. For example, C has a very flexible and lax memory access scheme using *pointers*, while the Java programming language imposes strict rules on memory management using *references*. So, in a conversion from C to Java, two paradigm shifts have to be made.

Demaine [5] developed a general method for converting C pointers to Java references. His theoretical considerations are well-founded and plausible, but unfortunately, he does not provide an implementation to show the feasibility of his approach. The technique for mapping pointers to references used in our Ephedra tool is similar to his approach.

2.3. C and Java Integration Strategies

From an integration perspective, 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 [16] (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.

To integrate C code into Java programs, these two hardware platforms have to be reconciled. Two main approaches exist: on the one hand, the Java Virtual Machine can be extended using code compiled to the native machine language of the target system, using the *Java Native Interface* (JNI) [15]. If the code is to be executed on several different target systems, it has to be compiled and installed on all of these systems. This may be acceptable for server applications that run on only one or few particular platforms, but is not suitable for client applications that usually need to run on a wide variety of different platforms. We therefore deemed it impractical for our purposes [22].

On the other hand, the C code can be compiled for the Java Virtual Machine, either directly or by first converting the C source code to Java. We surveyed several automated tools supporting this and found the following deficiencies [22]:

C2J++ [14, 31], a C++ to Java source transliteration tool, works well only for C++ code written according to a strict coding standard. In most cases a software engineer has to make extensive modifications to the generated Java code particularly if pointers are used in the original code.

C2J [25], a C to Java source transliteration tool, is able to transliterate large volumes of C code correctly to Java. However, the generated Java code is difficult to read and uses its own object model and storage allocation scheme that circumvents Java's type and run-time checking systems and makes it hard to interface the code with mainstream Java programs.

Java Backend for GCC [32], a C to Java byte code compiler, uses an object model and storage allocation scheme similar to that of C2J and has the same disadvantages.

3. The Ephedra Software Migration Environment

The previous section mentioned limitations and problems of these 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 compromising Java's type safety and security, together with possibly significant performance overheads.

The goal of our Ephedra migration environment is to supply a better solution to the problem of integrating C source code into Java programs. It provides a structured approach to migrating C source code to the Java Virtual Machine, minimising manual intervention by the software engineer wherever possible and guiding him or her wherever full automation cannot be achieved. The resulting Java source code does not circumvent the safety features of the Java Virtual Machine and can be easily integrated with existing Java programs. While the emphasis of our approach is on the C language, Ephedra also supports the conversion of the most commonly used C++ language elements.

3.1. Overview: Three Step Approach

Ephedra suggests a three step approach for the conversion of C and C++ programs to Java. The first step is necessary only during the conversion of K&R style C code. This kind of code does not contain function prototypes, thus limiting the ability of the compiler to perform type checking. To find and remedy major type inconsistencies, that may even be the cause of problems in the current code, all necessary function prototypes are inserted in the code. An ANSI C or C++ compiler is able to compile the code after this step, and test suites can be run to verify the correctness of the code.

Being a procedural language, C allows for data types to be explicitly related through composition only. Object-oriented languages also support inheritance relationships. In Step 2, data types and type conversions are analysed to find relationships that are implicitly present in the code and can be better and explicitly expressed using inheritance relationships. The code is then changed accordingly, and as a side effect, most Java incompatible type casts disappear from the code. This step will have to be performed on most C and many early C++ programs. If the subject system is written in C++ and uses multiple inheritance, it will also be converted to use only single inheritance at this time. After these transformations have been performed, the code can be compiled with a C++ compiler and verified using existing test suites.

In Step 3, the C/C++ source code is transliterated to Java source code. Data structures are converted to classes and

functions are converted to methods and collected in classes. During the transliteration, the code is also analysed for invocations of C storage allocation library functions, which are converted to near equivalents in the Java language. At the end of Step 3, the generated Java code can be compiled with any Java compiler and verified with the existing test suites.

The following sections describe these three steps in more detail.

3.2. Step 1: Insertion of C function prototypes

There are two major styles of C source code: Kernighan and Ritchie [12] first designed the C programming language in 1978, and source code written in that style is often called *K&R C*. In 1989, many developments and extensions to K&R C were standardised by the American National Standards Institute [1]. Code following this standard is usually called *ANSI C*.

The most obvious difference, and for Ephedra the most significant difference between these styles, is the use of *function prototypes*. In K&R C, only variables have to be declared before their use. Any unknown identifier that is followed by the function call operator is assumed to be the identifier of a function, and that function is assumed to take a variable number of arguments and return an integer value. It is left to the programmer to make sure that these assumptions are actually valid.

In ANSI C however, functions have to be declared using function prototypes that specify the return type and the number and types of the arguments, if any. These function prototypes have to be added to each source file before the first invocation of that function in the source file. Figures 1 and 2 show the differences between the two styles by means of a small example program.

As the consistency checking between function definitions and function invocations is left to the programmer in K&R C, there is a risk for introducing errors. To increase the chances for a successful migration from C to Java, it is important to eliminate sources of errors first. Therefore, as a first step of the Ephedra migration approach, a C program written in K&R style is converted to ANSI C style. The Free Software Foundation's GNU Compiler Collection (GCC [7]) comes with the tool *protoize*, which automates most of this conversion.

To improve the readability and correctness of the source code further, the return types of functions should be checked. If no return type is specified in a prototype of a function, a C compiler assumes it to be an integer. In many programs, this default formal return type is used even for functions that do not return a value. Modern C and C++ compilers can detect and flag these discrepancies and aid the software engineer in locating the code that needs change.

```
main(argc, argv)
int argc;
char *argv[]; {
int argument;

if (argc != 2) {
printf("usage: %s value\n",
argv[0]);
exit(1);
}

argument = atoi(argv[1]);
printf("the square of %d is %d\n",
argument, square(argument));
exit(0);
}

square(i)
int i; {
return i * i;
}
```

Figure 1. K & R style C program

Figures 3 and 4 show an annotated example.

3.3. Step 2: Data type and type cast analysis

Object-oriented programming languages greatly facilitate reuse by providing generalisation or specialisation by means of inheritance relationships. Canfora, *et al.* developed strategies for identifying classes within a system [3, 17, 18, 35, 8, 9, 13], but they usually do not take advantage of generalisation or specialisation features provided by the target programming language. It seems worthwhile to examine how commonalities and relationships of structures in a legacy system can be exploited to design inheritance hierarchies for the target system.

Type casts between otherwise unrelated structures in a C program are often used to model generalisation or specialisation features in procedural code [21]. These type casts pose problems, since type casts between data types that are not related through inheritance, are prohibited in Java. Even stricter type checking is performed at run time to prevent objects from being cast to a class type they do not actually instantiate.

To migrate a C program to Java successfully, either such type casts have to be removed, or the data structures involved in the type casts have to be changed to make their implicit relationship explicit. In [21], we presented methods, algorithms, and automated tools to identify problematic type casts and their associated data structures and to change these

```

// declare printf library function
#include <stdio.h>

// declare atoi and
// exit library functions
#include <stdlib.h>

// declare the square function
int square(int i);

// declares the main function
int main(int argc, char **argv)
{
    int argument;

    if (argc != 2) {
        printf("usage: %s value\n",
            argv[0]);
        exit(1);
    }

    argument = atoi(argv[1]);
    printf("the square of %d is %d\n",
        argument, square(argument));
    exit(0);
}

int square(int i)
{
    return i * i;
}

```

Figure 2. ANSI style C program

data structures so that their type casts are admissible for a Java program.

Another obstacle in the conversion process is the use of C++ multiple inheritance, which is only partially supported by specific language constructs in Java. Wen [33] examined the use of multiple inheritance in C++ programs and identified different categories of multiple inheritance. She also developed approaches to convert these different multiple inheritance types to Java using Java language constructs such as interfaces.

3.4. Step 3: Transliteration of source code

In the third and final step, the C source code, that was improved and corrected in Steps 1 and 2, is transliterated to Java source code. As one of our primary goals is the integration of the generated code into mainstream Java programs, readability of the generated code and compliance with the

```

// return type defaults to 'int'
// but no value is actually returned
foo() {
}

bar() {
    // statement is valid,
    // but result is indeterministic
    int i = foo();
}

```

Figure 3. Hidden error in C program

```

// state explicitly that
// there is no return value
void foo() {
}

void bar() {
    // C compiler correctly flags
    // this statement as error
    int i = foo();
}

```

Figure 4. Error turned explicit

Java object model are critical issues. Ephedra provides a tool to automate this transliteration completely.

The tool follows a four pass approach: initially, the source code is preprocessed, parsed, and type analysed. The IBM VisualAge C++ compiler is used as a front-end to perform these tasks. In Pass 2, the tool traverses the *abstract semantic graph* (ASG) generated in Pass 1 and creates a corresponding ASG that reflects the semantics of Java. Since C and Java are quite similar, many parts of the graph can be trivially transformed.

More complicated transformations are needed for the conversion of structured type declarations, initialisers, and expressions involving pointers. C2J [25] and the Java back-end for GCC [32] embed all structures and variables in a large integer array. This makes pointer operations and arithmetic easy and intuitive, but accessing the data through mainstream Java programs becomes difficult, and the data can be easily corrupted if there is an error in the code. Ephedra takes a different approach and maps C structured data types to corresponding Java classes. Where necessary, pointer arithmetic is *emulated* through member functions of these Java classes. This is accomplished without circumventing Java's storage allocation and memory protection schemes. Errors in the code will likely result in Java runtime exceptions and can thus easily be identified. A detailed description of all these transformations can be found

in the first author’s dissertation [20].

In Pass 3, the Ephedra tool traverses the newly created Java ASG. It determines which of the primitive variables in the original program need to be wrapped into classes so they can be passed and assigned by reference and then makes the necessary changes to the ASG.

Pass 4 is responsible for the generation of the Java source code from the Java ASG. Directories for packages, which are derived from namespaces in the C code, are created, and every class is written into one source file as required by Java. The tool also creates a *Makefile*, which helps to compile and run the generated code in the Ephedra environment.

4. Case Studies

To validate the Ephedra approach, we conducted three case studies by selecting typical C sources for migration to Java. To provide a usable migration environment, Ephedra offers automatic migration of frequently used C standard library functions. Many of them can be implemented quite easily and effectively using similar methods from the Java APIs. Some of them however are rather complex (e.g., the `fprintf()` function) and we decided to transliterate them from an existing C implementation rather than writing them by hand in Java. This transliteration served as the first case study.

For the second case study, we selected a small stand-alone game program that is in many aspects similar to the programs we intend to transliterate using Ephedra. It is written in K&R style C, uses pointers, function pointers, and `goto` statements, and contains little documentation.

We also selected two CPU intensive implementations of graph layout algorithms for the third case study to evaluate in part the performance of the generated Java code.

4.1. C standard library function

While looking for an existing, open source implementation of `fprintf()`, we found an ANSI C implementation, approximately 1,000 lines of C source code, written by Eberhard Mattes for his DOS and OS/2 port of GCC *emx* [23]. The implementation consists of one primary function and several subroutines. As the source code is already written in ANSI C, Step 1 of the Ephedra approach was not necessary. Analysis of the code showed further that it contained no problematic type casts, thus Step 2 could be omitted as well.

4.1.1. Step 3: Transliteration of source code

A first attempt to transliterate the code in Step 3 failed. The IBM VisualAge C++ front-end could not compile the source

code since it depended on some external functions, mostly for integer to string conversions, defined by *emx*-specific header files. We identified these functions and their purpose and decided to implement them as stubs first and to replace calls to them after the transliteration by calls to similar methods of the Java APIs.

The second transliteration attempt produced a syntactically correct Java class. Inspection of the Java sources showed that there were many casts between integer and boolean values, making the code quite difficult to read. Apparently, some of the variables and function return values used in the source assumed only boolean values, even though, because of the lack of a boolean type in C, their actual type was `char`. To optimise the code, we identified all of these variables in the original C source code and changed their type to `bool`.

After the third transliteration using Ephedra, we obtained a well readable and syntactically correct Java class. After replacing the calls to stubs created after the first transliteration attempt by calls to similar Java API methods, we were able to verify the correctness of the code using a few regression tests.

4.1.2. Evaluation and Optimisation

Even though our new Java implementation of `fprintf()` was now working correctly, the readability of the Java source code was not perfect. `fprintf()` returns the total number of characters written as its result, and so it has to check for I/O errors in various places to guarantee that its return value is exact. Most of the subroutines signal through their return values whether an I/O error occurred during their execution, and thus these return values have to be checked as well. These checks hide the algorithm behind the implementation and make the code difficult to read. In the original C source, these checks had been hidden using macros.

As file I/O is an ideal scenario for the use of exceptions, the Java APIs use them extensively for file access. We decided to modify the generated Java code to use exceptions to handle I/O errors. Since we do not know of existing tools to automate this process and we do not yet have sufficient experience with case studies to provide a general solution to this problem, we performed these modifications manually. In the end, we were able to replace all checks for I/O errors by a single `try/catch` statement in the primary `fprintf()` function. Moreover, the return type of all subroutines could be changed from `bool` to `void`. Through these modifications, the code size was reduced by about 25%.

The result of the transliteration process exceeded our expectations. Not only did we achieve the goal of obtaining a functional Java version of `fprintf()`, we also completed

the task quickly, and the resulting Java code was more readable than the original C code. Neither did we have to derive an algorithm from the specification of the `fprintf()` function nor did we have to reengineer it from the C implementation. There were no bugs, since the original C code had been well tested and all transformations performed were deterministic.

We do however have to note a problem with the optimisation step. Since this step is performed manually, it has to be repeated whenever the Java code is regenerated from the original C code. As we were dealing with C code that does not change any more, this was not a problem in this case study. When we are dealing with original code that still changes, we have to be careful to record all optimisations so they can be re-applied whenever the Java code is regenerated.

4.2. Stand-alone C program

During the development of the Ephedra source code transliteration tool, we wrote various small C programs as a regression test suite to test certain aspects of the transliteration. To validate the correctness of the many applied transformations in the context of a real program that uses a large range of C language features, we chose a stand-alone *monopoly* game program, which we had used in previous studies [27, 28]. This program is written in K&R style C code and uses complex data structures, pointers, and function pointers. As such, it constitutes a solid test case for Step 1 and the more difficult transformations of Step 3 of the Ephedra approach.

4.2.1. Step 1: Insertion of C function prototypes

In a preparatory step, we compiled and ran the original monopoly program to get an impression of its functionality. According to Step 1 of the Ephedra approach, we then used the *GNU protoize* tool to transform the program from K&R style C to ANSI C. The monopoly program was then ready to be imported into and compiled in the IBM VisualAge C++ IDE. The first attempt to compile the program failed — the C++ compiler detected some errors that the C compiler had been unable to discover due to the lack of prototypes in the K&R style C program. Using the browsing capabilities of the IBM VisualAge C++ IDE, detection of the source of the errors was easy.

In some cases, extra parameters were passed to a function (Figure 5). We determined that these extra parameters were indeed not needed. We had the choice of either correcting all invocations of the function in question, or adding an extra default parameter to the function prototype that would be ignored in the function body. We decided for the latter to minimise source code changes (Figure 6). In retrospect, we should have chosen the other option. The addition

```
/*
 * This routine executes a truncated
 * set of commands until a "yes or
 * "no" answer is gotten.
 */
getyn(char *prompt) {
    [...]
}

[...]
// correct invocation;
getyn("enter yes or no");

// compiler detects mismatch
char *yn[];
getyn("enter yes or no", yn)
```

Figure 5. Error in Monopoly Program

```
/*
 * This routine executes a truncated
 * set of commands until a "yes or
 * "no" answer is gotten.
 */
getyn(char *prompt, void* = 0) {
    [...]
}

[...]
// both invocations now correct
getyn("enter yes or no");
char *yn[];
getyn("enter yes or no", yn)
```

Figure 6. Corrected Example Code

of the superfluous parameter did not correct the errors in the code, but rather obscured them and might also confuse future developers working on the code.

The compiler also flagged discrepancies between declared return types and actual return values of functions. None of the functions in the code had a return type explicitly specified, and thus they all defaulted to `int`. Some of the functions did not actually return a value and were flagged as incorrect by the compiler. We corrected them by giving them an explicit return type of `void`.

The compiler detected more problems in the functions responsible for loading and saving games. Apparently, the code used some Unix system calls to save and load its data area. Since we used the Windows version of the IBM VisualAge C++ compiler, these system calls were unknown to the compiler. After looking at the code we found it dif-

difficult to understand and decided that it would be better to re-implement the load and save functions from scratch in Java rather than to emulate the Unix system calls used by their current implementations. We therefore replaced these functions with stubs. The code now compiled without errors and Step 1 of the Ephedra migration method was complete.

4.2.2. Step 2: Data type and type cast analysis

The automated tool to detect problematic type casts for Step 2 did not locate any such casts. Since the program was written in C rather than C++, we did not have to transform any instances of multiple inheritance.

4.2.3. Step 3: Transliteration of source code

The Ephedra source code transformation tool performed most of the work necessary for Step 3. The tool does not yet transform `goto` statements, and, hence, these constructs had to be converted manually. The remaining transformations were executed automatically. The generated Java source code then compiled correctly.

This was our first comprehensive test of the transformation tool. Unfortunately, the resulting program did not work properly at first. Some of the library functions had to be implemented or corrected and some of the code transformations had been implemented incorrectly. These errors usually resulted in an exception to be thrown exactly at or very near the point in the code where the actual error was located. It was therefore easy to locate and correct the errors in the library functions or the transformation tool. The Java code was then re-generated from the C code and compiled and tested again until a correct version of the program was obtained.

4.2.4. Evaluation

To obtain a better estimate of the time needed to perform a complete conversion, we repeated the entire transformation process. The process was completed in a few hours by a different developer who is familiar with moving programs from K&R C to C++. Thus the time it took to convert this game program is considerably less than if we had to rewrite the program in Java. The number of lines of code is approximately the same for the original C program and the resulting Java program.

In a final step, we re-implemented the load and save functions that we had removed earlier in Java. The `java.io.Serializable` interface helped to make this implementation simple.

4.3. Conversion of two CPU intensive algorithms

To evaluate not only the correctness but also the efficiency of the generated code, we chose to convert two CPU intensive programs. These programs are part of *sgraph* [11, 2] and implement *Spring* and *Sugiyama* graph layout algorithms. The programs work as filters (i.e., they read their input from standard input and write the results to standard output). The graphs they operate on are stored in a proprietary graph format. Lex and Yacc are used to parse this format.

As the graph format is rather simple and we did not want to transform Lex or Yacc in this case study, we decided to re-implement the parser without Lex and Yacc for testing purposes. This new parser was implemented in about 100 lines of C code.

The Rigi graph editor [24, 26] uses the Spring and Sugiyama stand-alone programs for generating graph layouts. As the layout programs did not come with test suites, we used this graph editor to generate a few test cases.

4.3.1. Step 1: Insertion of C function prototypes

As the programs were written in K&R style C, we used GNU protoize to transform them to ANSI C. While compiling the programs using the VisualAge C++ IDE, the compiler found a few problems. As in the monopoly program, the compiler noticed discrepancies between declared return types and actual return values of functions. Some of the functions in the code had no return type explicitly specified, and thus their return types defaulted to `int`. As they did not actually return a value, they were flagged as incorrect by the compiler. We corrected them by giving them an explicit return type of `void`. The compiler also encountered an incorrect function call: an extra parameter had been specified.

4.3.2. Step 2: Data type and type cast analysis

The *sgraph* data structures used a *union* to save attributes of the graph. These unions could contain either one integer or an untyped pointer. According to Step 2 of the Ephedra approach, this union should be converted to a hierarchy of classes with an abstract base class and concrete subclasses for the different types of attributes.

Since the fields of the union were rather small, we decided to take a different approach and transform the union to a regular data structure with one field each for the integer and pointer value. We felt that the little extra amount of memory used in this approach was preferable to the loss in performance that would result from type casts or calls to virtual functions when strictly following the Ephedra approach.

The source code transliteration tool (Step 3) performs this transformation automatically for all unions that are still present in the code. Thus, we did not have to make any manual changes to the code.

4.3.3. Step 3: Transliteration of source code

The transliteration of the code went smoothly, and no major problems were encountered. The size of the generated source code was about the same as that of the original source code.

The programs used some C standard library functions, which had not been implemented yet. Once they had been implemented, the Spring layout program performed correctly. There were minor differences in the output of the original and transformed program. They were caused by differences in how C and Java handle floating point numbers.

The Sugiyama layout program aborted with a null pointer exception while printing the results. Investigating the problem revealed a severe error in the original C program: some memory was released before it was last used. As the program did not allocate any new dynamic storage after this release of storage, the storage was not overwritten and the C program still functioned properly. In the Java transliteration, the de-allocation of memory is emulated by setting references to storage that is to be de-allocated to `null`. A subsequent access to those references thus resulted in a null pointer exception.

4.4 Performance Evaluation

For small input files, the transformed versions of the Spring and Sugiyama graph layout algorithms performed worse than the original program. It was quickly determined that this is mainly due to the long start-up time of Java programs.

For larger examples, the difference in execution time becomes smaller, but is still significant. For a graph with 560 nodes, the transformed version of Spring takes about five to seven times as long as the original version, depending on the Java Virtual Machine used. Sugiyama performs worse. Laying out a graph of 270 nodes with the Java version of the program takes twenty to forty times longer than with the C version. Examination of the problem showed that the transformed programs use huge amounts of memory for arrays of integers: instead of 2 MB, Sugiyama uses 80 to 160 MB of memory, depending on the virtual machine. Obviously, there is room for improvement both in the transformation performed by Ephedra and in the Java Virtual Machine.

These results prompt us to reconsider the techniques we currently use for transforming pointers and arrays. We will have to develop a strategy to allocate storage in a way that

does not put such high demands on the Java Virtual Machine. If this new strategy compromises other goals of the transformation process, we will have to let the developer prioritise and select transformation strategies.

5. Conclusions and Future Work

Since electronic commerce over the Internet plays a prominent role in today's business environment, customers expect integrated Web-based services historically provided by legacy information systems. One way to provide these services to the users quickly and effectively is to integrate the legacy C code of such information systems with newly developed Java programs. In this paper, we presented several integration strategies by discussing their characteristics, strengths, and weaknesses. We also introduced the Ephedra approach for migrating C programs to Java using the Ephedra software migration environment. Three different migration case studies were used to evaluate this approach to migrating C/C++ code to Java.

Two of the case studies showed that debugging the generated code seems to be easier than debugging C code compiled to native machine code. The run-time checks performed by the Java Virtual Machine help greatly in locating faults that may go undetected in C programs under certain conditions. It may be possible to decrease development time and cost for new C/C++ programs by transliterating them to Java and debugging them using the Java Virtual Machine. We are pleased with the Java code generated by the transliteration tool. The generated code seems to be correct and reasonably efficient. If certain guidelines are followed in the development of new C++ code (e.g., portable type casts, single inheritance), no manual intervention will be required to migrate such C++ code to Java. Moreover, it is quite likely that faults detected in the Java code are faults in the original C code. Debuggers could be instrumented to hide the intermediate Java code from the developer and make it appear as if he or she was debugging the original C/C++ code. A study comparing the development cost when using this approach, as opposed to the cost of traditional development, might be interesting. This might be a viable and useful option for C development environments. We did not anticipate this application when we began this project.

The monopoly application was used in previous case studies [27, 28] to evaluate the usability of reengineering tools in a structured experiment. This experiment could be repeated to determine whether the source code migration helps or hinders the software engineer in understanding the program. Such studies would also help us to evaluate how readable and maintainable the generated code is. As it is difficult and not very objective to judge the readability of code by simply looking at the code, we have consciously avoided

conclusions about the code readability in this paper. Since the authors of the paper are familiar with the conversions applied in these case studies, they probably rate the code as more readable compared to Java developers who are not familiar with the code.

Some parts of Ephedra (e.g., goto conversion) could be further automated or improved to speed up the transformation process and to relieve the software engineer. The automated tools that are currently provided do not have a uniform user interface. The IBM VisualAge C++ IDE could be extended to provide presentation integration for all the automated tools. Finally, the transformation tool should preserve the comments in the code. This involves finding a method to identify to which token a particular comment in the source code really refers to.

Step 2 of the Ephedra approach (i.e., data type and type cast analysis) has so far only been tested with specially constructed examples [6]. It was not needed in the case studies reported in this paper. We will have to apply the entire Ephedra method to a program that requires all three Ephedra steps to demonstrate their effective integration and interaction.

While these case studies were very successful, we need to conduct more case studies with larger programs to validate the Ephedra approach further. In particular, we would like to evaluate the integration of the generated Java code into mainstream Java programs.

References

- [1] American National Standard X3.159-1989.
- [2] Sgraph Standalone Programs. <http://www.infosun.fmi.uni-passau.de/GraphEd/download/sgraph-standalone/>, December 2001.
- [3] G. Canfora, A. Cimitile, and M. C. Munro. An improved algorithm for identifying reusable objects in code. *Software Practice and Experiences*, 26(1):24–48, 1996.
- [4] A. Cimitile, A. D. Lucia, G. A. Di Lucca, and A. R. Fasolino. Identifying Objects in Legacy Systems. pages 138–147, May 1997.
- [5] E. D. Demaine. Converting C Pointers to References. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, CA, 1998.
- [6] H. Erdogmus and O. Tanir, editors. *Advances in Software Engineering: Topics in Comprehension, Evolution, and Evaluation*. Springer-Verlag New York, Inc., Dec. 2001.
- [7] Free Software Foundation (FSF). GCC Home Page. <http://www.gnu.org/software/gcc/gcc.html>, October 2001.
- [8] H. Gall and R. R. Klösch. Finding objects in procedural programs: an alternative approach. In *Proceedings of 2nd IEEE Working Conference on Reverse Engineering*, pages 208–216, Toronto, Canada, July 1995. IEEE Computer Society Press.
- [9] J. George and B. D. Carter. A strategy for mapping from function-oriented software models to object-oriented software models. *ACM Software Engineering Notes*, 21(2):56–63, Mar. 1996.
- [10] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., 1996.
- [11] M. Himsolt. Sgraph Programmer’s Manual, 1993.
- [12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.
- [13] 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.
- [14] C. Laffra. C2J, a C++ to Java translator. (web site no longer available).
- [15] S. Liang. *The Java Native Interface, Programmer’s Guide and Specification*. Addison-Wesley Longman, Inc., 1999.
- [16] F. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Inc., 1999.
- [17] S.-S. Liu and N. Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. In *Proceedings of IEEE Conference on Software Maintenance*, pages 266–271, San Diego, CA, Nov. 1990. IEEE Computer Society Press.
- [18] P. E. Livadas and T. Johnson. A new approach to finding objects in programs. *Journal of Software Maintenance: Research and Practice*, 6:249–290, 1994.
- [19] A. J. Malton. The Migration Barbell. First ASERC Workshop on Software Architecture, Aug. 2001. <http://www.cs.ualberta.ca/~kenw/conf/awsa2001/papers/malton.pdf>, November 2001.
- [20] 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>.
- [21] J. Martin and H. A. Müller. *Discovering Implicit Inheritance Relations in Non Object-Oriented Code*, chapter 11. In Erdogmus and Tanir [6], Dec. 2001.
- [22] J. Martin and H. A. Müller. Strategies for Migration from C to Java. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 200–209, Lisbon, Portugal, Mar. 2001.
- [23] E. Mattes. emx 0.9d (GCC and tools for DOS & OS/2). <http://archiv.leo.org/pub/comp/os/os2/leo/gnu/emx+gcc/>, October 2001.
- [24] H. A. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988.
- [25] Novosoft. C2J — C to Java translator. <http://www.novosoft-us.com/NS2B.nsf/w1/C2J>, September 2001.
- [26] U. of Victoria. Rigi Web Server. <http://www.rigi.csc.uvic.ca>, June 2001.
- [27] M.-A. D. Storey, K. Wong, P. Fong, D. S. Hooper, K. Hopkins, and H. A. Müller. On Designing an Experiment to Evaluate a Reverse Engineering Tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering*, Monterey, CA, Nov. 1996.
- [28] M.-A. D. Storey, K. Wong, and H. A. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proceedings of the Seventh Working*

- Conference on Reverse Engineering*, pages 12–21, Amsterdam, Holland, Oct. 1997.
- [29] A. A. Terekhov. Automating Language Conversion: A Case Study. In *Proceedings of the International Conference on Software Maintenance (ICSM) 2001*, pages 654–658, Florence, Italy, Nov. 2001.
 - [30] A. A. Terekhov and C. Verhoef. The Realities of Language Conversions. *IEEE Software*, pages 111–124, Nov. 2000.
 - [31] I. Tilevich. Translating C++ to Java. *First German Java Developers' Conference Journal*. <http://sol.pace.edu/~tilevich/c2j.html>.
 - [32] T. Waddington. Java Backend for GCC. <http://archive.csee.uq.edu.au/~csmweb/uqbt.html#gcc-jvm>, November 2000.
 - [33] T. Wen. Translating C++ to Java: Resolution of Multiple Inheritance. Master's thesis, University of Victoria, 2000.
 - [34] 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.
 - [35] A. S. Yeh, D. R. Harris, and H. B. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Proceedings of 2nd IEEE Working Conference on Reverse Engineering*, pages 227–236, Toronto, Canada, July 1995. IEEE Computer Society Press.