Convert2Java:
Automatic Conversion of C to Java

V.H. Allan and X. Chen
Utah State University
Computer Science Department
Logan, UT 84322
allanv@cs.usu.edu

Automatically converting code from C to Java can be successfully done. This paper presents the techniques of a C to Java converter, Convert2Java, developed at Utah State University. The C to Java converter is tested on programs from a popular data structures text. In our translator, ease of maintenance of the target code is treated as a more important goal than identical results of execution. Because of the need to create logical classes from procedural code, translation has three phases: (1) evaluation of the suitability of automatic translation (2) division of the source code into logical classes (3) translation of each class separately. Results are encouraging. Over 90% of the examples are translated with no changes or only minor modifications required.

Key words: automatic translation, C to Java conversion, pointers in Java

1 Introduction

Hundreds of millions of lines of legacy code are written in a variety of languages. That code will not be replaced quickly [GFKH99]. Automatic translation of legacy code to a more portable language is one option. There are several considerations in the decision to port a system from C to Java. These include portability, maintenance, seamless conversion, efficiency, and cost. The relative importance of these factors dictates which course to follow in performing the change of language. There are several alternatives: invoke an automatic translation system, redesign the project using Java, or put a Java wrapper around the C code, using the original code as *native code*.

A Java wrapper around C code has the advantages of little coding effort and minimal differences between the original system and the Java system, but it becomes awkward to maintain and port to various systems. A redesign produces the best Java code but is very expensive and time consuming. An automatic translation system has the advantage of being cost effective, yet requires testing to be sure no subtle differences exist between the two systems. If the resultant code is maintainable, it becomes a viable alternative. Our approach in creating Convert2Java has the advantage of producing code with excellent readability. The user will choose between the various translation choices for a particular piece of code based on prior evaluation of automatic translation.

In this paper, we consider translation issues of an automatic system, Convert2Java. Because our interest lies in producing code which can be maintained, our goal is focused on preserving the logical structure of the code. There are several phases: (1) determining whether a given piece of code is a good candidate for automatic translation, (2) dividing the source code into logical classes and (3) actually performing the translation of each logical class. This paper focuses on the process of automatic translation of code into a single class.

Because we are converting from a procedural language to an object oriented language, we need to associate data and functions with a class. We use graph algorithms to perform graph partitioning based on cohesion [HL95] which determines which data and functions logically belong together. Using a bipartite graph in which nodes represent either data or functions and arcs are annotated with the strength of the association, we are able to determine the logical association and the costs of various divisions into classes.

The goal is to convert the C code to Java in such a way as to make maintenance of the target code viable. We do not want to be forced to change the original C code and retranslate each time changes are required. The Java code resulting from the translation will probably be slower that the original C code [JA99, PTB+97], but this decrease in speed should be known before translation is attempted. Because the process of translation could uncover subtle differences,[1] the resultant code will need to be tested. Some programs lend themselves to rigorous testing, while others do not lend themselves to a feeling of confidence. If automatically translated code will be difficult to prove correct, this negatively impacts the desirability of selecting this alternative.

It would be helpful to determine beforehand the degree to which automatic translation will be effective, in terms of correctness, maintainability, testability, and efficiency of resultant code. At times the translated file may be awkward, making subsequent maintenance more difficult. The goal of our variant of quality prediction [CW94] is to determine which programs are good candidates for automatic translation and which are not. In this paper, we assume that cohesiveness testing has been performed and that the code in a physical file belongs to the same class. We also assume that the code is a viable candidate for automatic translation.

2 Overview

We use JavaCC, the compiler writer tool released with Sun's JDK, as the basis for the translation. We assume that the C code to be translated is valid. Although many invalid constructs will be

[1] Expected differences include those resulting from different sizes of base types, data operations depending on a specific bit representation, interaction between union elements or actions unspecified by the C language which vary between implementations, such as evaluation order or exception handling.

Original C Code	**Corresponding Java Code**

`int global_a;`
`void foo(int t) { ... }`
	`class test1`
	`{`
	`static int global_a = 0;`
	`static void foo(int t) { ... }`
	`}`

Figure 1: Converting C code to a class.

flagged as such, a superset of the legal C constructs is accepted by the compiler. Many language features in C have a direct counterpart in Java [PTB+97], making translation straight forward. This paper concentrates on the aspects of C code for which translation is not obvious. Each C file is translated into an equivalent Java file of the same name. The code of Figure 1 illustrates the conversion.

3 Customization

We provide a convenient way of mapping functions from C to Java. A library function in C may have a different name in Java, may not exist at all, or may have several variants. The funmap.h file is a way for the user to tell the compiler how he wants function names translated. For example, the C library function *exit* can be translated as *System.exit* in java. Thus, the funmap.h file would contain the lines

```
$def exit    System.exit
void exit(int);
```

The $def line means that all instances of `exit` in the C file will be replaced with `System.exit` in the translated Java code. The prototype is included so the the compiler can add the function to the symbol table (rather than have to locate the prototype in the header file).

As another example, there is no counterpart to `fscanf` in Java. Thus, we will need to either translate the source code for `fscanf` (using our compiler) or write the code by hand. Once we have the code for `fscanf`, we will place it in a class, say `mystdio`. Then, the funmap file will contain the lines

```
$def fscanf   mystdio.fscanf
int fscanf(FILE *, char *, ...);
```

This gives us tremendous power. First, we translate needed libraries from their source code. Then, we alert the compiler to the location of the libraries via a pair of funmap entries. Finally, we compile the program which uses the library routines.

4 Type conversion

Because of the inherent differences in the languages, one of the most difficult translation issues is that of type conversion. Java has many features to make bug detection easier. It is impossible to assign one object to another if the types are incompatible [vH97]. Other concerns involve the translation of a C type to the most appropriate Java type.

Simple Types

C does not outline the kind of floating-point arithmetic used to do simple arithmetic [vH97]. Thus, we expect some differences in results after porting to Java. Some types predefined in C are not supported in Java. There is no unsigned type in Java. Hence, both signed and unsigned C types `int, char, long, float,` and `double` are mapped to the Java type of the same name. All variants of the `long` C data type are mapped to the Java type `long`.

Structs and Unions

Structures are translated directly to classes with public member variables. However, unions pose a more difficult problem. In Java, there is no concept of a union in which the same space is associated with different data types simultaneously, as such practice is seen as a violation of strong type checking. In order to determine a reasonable translation, we need to look at the reasons for using a union. In C, unions are used for two reasons: (a) to conserve space by reusing the space for a different purpose at disjoint intervals of time, or (2) wanting to view the same space in a different way simultaneously.

With Java's garbage collection, conserving space is not a concern; it is easy to discard space and reuse it later. Thus, implementing the code to effect a true union serves no purpose in this case. We relax the requirement that the implementation is physically equivalent to the original C storage and treat it as a struct instead of a true union.

However, the second case (needing to view the same space in different ways simultaneously) is more difficult to handle. Because the internal representation of data in the source and target language is different, it is not clear that anyone wanting to view the space differently would get equivalent results under Java. Unions used in this manner will not be translated correctly. A solution would be to apply data flow techniques and live track analysis to determine if such unsupported use of unions is present.

Original C Code

```c
struct Shape
{
  int type;
  union { char *v; int l; } u;
};
```

Corresponding Java Code

```java
public class Shape
{
  int type = 0;
  Tmp_struct_0 u = new Tmp_struct_0();
}


public class Tmp_struct_0
{
  charPointer v = new charPointer();
  int l = 0;
}
```

Figure 2: Converting C structs and unions to Java.

Original C Code

```c
enum fruit { apple, kiwi,
        grape, banana = 5, orange};
```

Corresponding Java Code

```java
public class fruit
{ public static final int apple = 0;
  public static final int kiwi = 1;
  public static final int grape = 2;
  public static final int banana = 5;
  public static final int orange = 6;
}
```

Figure 3: Implementing enumerated types in Java

Unions are implemented as a struct; with no sharing of space between components. This is shown in the example code of Figure 2, in which a class is introduced as a name for the union:

Enumerated Types

C allows user defined enumeration types, but Java does not. An enumerated type is translated into a separate class consisting only of static, final variables. The translator replaces all uses of values of the enumeration by access of a class data member in Java. Constant values are allowed.

Consider the example of Figure 3. A use such as **fruit favorite = apple;** becomes **int favorite = fruit.apple;**. This translation is a true translation of the original code (since `enums` are stored as integers and are not distinguished from `ints` in operations), but it has the result of blurring the distinction between a `fruit` and an int.

Boolean type

There is no standard boolean type in C. Integers which evaluate to zero are considered false. Any non-zero integer value is treated as a true. Because Java is strongly typed, integers are not allowed to be treated as booleans or vice versa.

An integer expression *exp* which is used as a boolean in C is translated to `(exp) != 0` in Java. Accordingly, an expression yielding a boolean will be translated into an expression of type *int*; a boolean expression, *bexp*, is translated as `(bexp)?1:0`. In the event of a `int` function which is used as a boolean, such as a `int isGood(...)`, a direct translation may both change the boolean expression contained in `isGood` to an `int` and then change the return value of `isGood` back to a `boolean` when it is used. Use-def chaining can be used to determine the proper translated type for integer source functions and eliminate such double conversion.

Pointers

The handling of pointers is probably the most interesting aspect of Convert2Java. C allows programmers to perform arithmetic on pointers. Although Java has references instead of pointers, it provides no way for the user to control when a data item is a reference and when it is a copy. C allows pointers to be "forged", an important omission in Java [vH97]. Simple variables are always passed by value; there is no way to return a simple value through the argument list. Class variables are always passed by reference; because no copy is made, there is no way to restrict changes to the local version of a passed parameter. In other words, simple variables are always passed using the *call by value* model and class variables are always passed using the *call by reference* model.

Pointers to structures can often be converted directly to class references. Because you only have references, if you wanted a copy made when you pass a structure as a parameter in C, you have to physically make a copy of the corresponding class in Java; there is no way to have a class which is not a reference. Because you only have references, you cannot take an address of an object nor can you have an address of an address. In order to pass back a value of a simple type through a parameter list, a simple type must be stored as a member variable within a class reference which is passed. This makes the code considerably more complicated. In addition, the distinction between a pointer and an array is blurred in C. In Java, an array type is distinct from a reference to that type.

Pointers to simple variables

A pointer to a simple variable has no direct translation in Java. In affect, for each pointer variable of base type X, we create an array of X and manage the memory ourselves. This special type is called an *intPointer, charPointer, bytePointer, floatPointer, or longPointer.* These pointer types

Original C Code

```
int a[5], b;

int *p, *q;

p = a;
b = *p;
p++;
q = p;
b = p[2];
b = *(q + 3);
q += 3;
q -= 4;
p[3] = q[2];
```

Corresponding Java Code

```
int[] a = new int[5];
int b = 0;
intPointer p = new intPointer();
intPointer q = new intPointer();
p.assignArray(a, 0);
b = p.getArray()[p.pos()];
p.inc(1);
q.assignArray(p.getArray(), p.pos());
b = p.getArray()[p.pos() + 2];
b = (q).getArray()[(q.pos() + 3)];
q.inc(3);
q.inc(-(4));
p.getArray()[p.pos() + 3] =
          q.getArray()[q.pos() + 2];
```

Figure 4: Handling of pointers

consist of a reference to the array and the current subscript to use in dereferencing a pointer type. If two pointers either point to the same location or point to different locations in the same C array, they will have a reference to the *same* Java array in the corresponding Pointer type.

Consider the example of Figure 4. For the statement `b = *p;`, we must first get the array associated with p (`p.getArray()`) and then subscript it with the current subscript (`p.pos()`). When we assign `p[3]=q[2]`, we must get the arrays associated with each variable and add the respective offset to them before assigning (`p.getArray()[p.pos() + 3] = q.getArray()[q.pos() + 2];`).

Whenever an address of a simple variable is taken, we must create a temporary pointer to implement the pointer operations. Whenever a reference to the variable is modified, the variable is updated accordingly. The symbol table keeps track of the current aliases. In the example of Figure 5, a temporary variable $\_p\_a$ is created as a reference to a, which is assigned to p immediately. p is then assigned to q. Both p and q are now pointers to a. When $*p$ or $*q$ is modified, a is also updated. p and q are virtual references to a.

Note how pointers are used to affect a swap in the code of Figure 6. In this example, we pass the temporary variables $\_p\_a$ and $\_p\_b$ to the function `swap`. Variable a and b are updated after the call to `swap`.

Original C Code

```c
a = 0;

int *p, *q;
p = &a;




q = p;
*p = 1;


*q = 2;
```

Corresponding Java Code

```java
int a = 0;
intPointer p = new intPointer();
intPointer q = new intPointer();
{
  intPointer _p_a =
      new intPointer();  // Pointer to a
  _p_a.assignValue(a);
  p.assignArray(_p_a.getArray(), 0);
}
q.assignArray(p.getArray(), p.pos());
{
  p.getArray()[p.pos()] = (int)(1);
  a = p.getValue();
}
{
    q.getArray()[q.pos()] = (int)(2);
    a = q.getValue();
}
```

Figure 5: Manipulating a reference.

Array

For a dynamically allocated array in C, one must declare a pointer and allocate memory to it at runtime. In Java, an array is a separate type and is dynamically allocated. A statically defined array in C is directly translated into an array object in Java with an initial size. Since it is not clear whether a C pointer to a simple type will be used as an array or a reference to a variable, we always treat it as a pointer rather than an array. The implementation of pointer can handle either use efficiently.

When a multi-dimensional array is defined, the form of the definition determines the actual coce produced. See the code of Figure 7.

Struct instances

When we define a variable of type struct in C, it represents a memory block with the size of the struct. If a pointer to the struct is defined and is assigned a memory block. The pointer contains the address of the memory block. In Java, each object has a unique identity, called a handle. When an object variable is first defined, it represents the handle to an object. An object should either be assigned a handle or contain a null value. An object variable in Java is equivalent to a pointer type variable in C if we regard the address of a memory block to be a unique identity for an object.

Original C Code

```
void swap(int *a, int *b)
{
    int temp;
    temp = *b;
    *b = *a;
    *a = temp;
}

void test()
{
    int a = 5, b = 6;

    swap(&a, &b);
}
```

Corresponding Java Code

```
static void swap(intPointer a, intPointer b)
{
    int temp = 0;
    temp = b.getArray()[b.pos()];
    b.getArray()[b.pos()]
        = a.getArray()[a.pos()];
    a.getArray()[a.pos()] = temp;
}

static void test()
{
    int a = 5;
    int b = 6;
    {
        intPointer _p_a = new intPointer();
        _p_a.assignValue(a);
        intPointer _p_b = new intPointer();
        _p_b.assignValue(b);
        swap(_p_a, _p_b);
        a = _p_a.getValue();
        b = _p_b.getValue();
    }
}
```

Figure 6: Passing addresses of simple types.

Original C Code

```
void foo(int k[][5])
{}


void goo (int *p)
{ int a;
  a = p[3];
}


void test()
{
  int a[5];
  int b[3][5];
  int * c[3];
  foo(b);
  goo(b[1]);
  goo(c[2]);
}
```

Corresponding Java Code

```
class multi
{
static void foo(int[][] k)
{}
static void goo(intPointer p)
{
    int a = 0;
    a = p.getArray()[p.pos() + 3];
}
static void test()
{
    int[] a = new int[5];
    int[][] b = new int[3][5];
    intPointer[] c = new intPointer[3];
    foo(b);
    goo(new intPointer(b[1], 0));
    goo(new intPointer(c[2].getArray(),
                        c[2].pos()));
}
}
```

Figure 7: Multi-dimensional arrays.

Because there is no distinction between an object instance and an object pointer in Java, we map both struct variables and struct pointers to object variables in Java. If a struct variable is passed as a parameter, we must make a copy of the referenced Java value so that changes made to its contents in the function are local and do not affect the passed argument. In the translation from a struct to a class, a default `copy` function is automatically generated to make this possible. If a struct variable is not passed to a function, there is nothing special that needs to be done to account for the fact that the instance is actually being implemented as a reference.

We still have the problem that a pointer to a struct may actually be an array of structs. In the case of a simple pointer, we declared a Pointer variable (with its associated array and current subscript) which was used regardless of the actual usage of the simple pointer. This results in considerable code complexity but was required to accommodate either possibility. However, for a pointer to a struct, the extra complexity is required for an array but is not required for a class reference (as no array is needed). We can use context to infer the correct type. Using data flow techniques and live track analysis, we can determine the best target type for a particular instance of a struct pointer.

5 Variable Length Argument List for a Function

A variable number of parameters is not supported in Java. We translate the variable list into an object of Java type `Vector`, which holds the various types of parameters. A Java Vector holds a variable number of variables. The type of each parameter in the Vector must be derived from Object. Thus, pointers are stored as the corresponding object in Java (since the base type is a descendant of Object). If the parameter is a simple type variable in C, it must be converted to an object of the simple type's wrapper class (which derives from Object where as a simple type does not). The code of Figure 8 illustrates this conversion.

6 Limits of Translation

Simple types

The number of bytes for various types are different in C and Java. For instance, a `char` in C is 1 byte, but it is 2 bytes in Java. Thus, the user must be careful when the code has bit operations which rely on a specific number of bits in the representation.

There is no unsigned type in Java, so all unsigned types are converted to the signed types. If the range of a variable is always positive, a C program may use `unsigned char` or `unsigned int` to increase the range of values than can be stored in the same space. In Java, both `char` and `int` have a much wider range than `unsigned char` and `unsigned int` in C, so eliminating the unsigned designation may not affect correctness. However, any computation which depends on the precise

Original C Code

```
int max( int how_many, ... );

main()
{int max_int, a;
 a=43;
 max_int = max( 1, a,24 );
 max_int = max( 1, 4,a,-5 );
 max_int = max( 1,a, a*a, 3,423);
 return 1;
}
```

Corresponding Java Code

```
class t
{
static int main()
{
    int max_int = 0;
    int a = 0;
    a = (int)(43);
    {
        global.args = new java.util.Vector();
        global.args.addElement(new Integer(a));
        global.args.addElement(new Long(24));
        max_int = max((int)(1), global.args);
    }
    {
        global.args = new java.util.Vector();
        global.args.addElement(new Long(4));
        global.args.addElement(new Integer(a));
        global.args.addElement(new Long(-5));
        max_int = max((int)(1), global.args);
    }
    {
        global.args = new java.util.Vector();
        global.args.addElement(new Integer(a));
        global.args.addElement(new Integer(a * a));
        global.args.addElement(new Long(3));
        global.args.addElement(new Long(423));
        max_int = max((int)(1), global.args);
    }
    return 1 ;
}
}
```

Figure 8: Converting C code to a class.

range of values which can be stored may be affected. Another reason for declaring a variable to be `unsigned` is for use with shift operations. However, the Java shift operation is similar to that expected for `unsigned int`.

Pointers to functions

We do not support the C construct of pointers to functions. In Java, we can effect passing of different functions to a method by having the desired function name overloaded as a member of each instance of a parameter class. Since this is a major design change, we do not attempt to make this modification automatically.

Explicit type conversion

The ability to perform explicit type conversion in a strongly typed language like Java is limited. We support explicit type conversions for simple types. We do not support conversion for pointers. For example, casting a “pointer to a struct” to a “character pointer” is useful in C, but impossible in Java.

Expressions which Are Non-repeatable

A non-repeatable expression (which cannot be repeated without altering the value) causes problems in automatic code generation. For example, in the original code of Figure 9, we can not use A[rand() % (j+1)] in the code that is produced without changing the results; the expression appears more than once (as shown in the right hand side of Figure 9), but does not evaluate to the same value each time. In order to correct this error, we need to modify the C code as shown in Figure 9.

Conditions which are translated into multiple statements

Sometimes an expression in C is translated into several statements. If this happens in a condition expression, the automatic conversion is not allowed. Such situations require manual intervention and are flagged during translation.

Original C Code

```
swap(&A[j], &A[rand()%(j+1)]);
```

***Need to modify as**

```
t=rand()%(j+1);
swap(&A[j], &A[t]);*\
```

Translation of Original C Code

```
intPointer _t_0 = new intPointer();
_t_0.assignValue(A.getArray()[A.pos()+j]);
intPointer _t_1 = new intPointer();
_t_1.assignValue(A.getArray()[A.pos()+
      stdlib.rand()%(j+1)]);
swap(_t_0, _t_1);
A.getArray()[A.pos()+j] = _t_0.getValue();
A.getArray()[A.pos()+stdlib.rand()%(j+1)]
       = _t_1.getValue();
```

Figure 9: Removing a self-modifying expression from a parameter.

7 Results and Conclusions

We tested the converted by running programs of at least 50 lines in length from various programming and data structures books. These programs were selected because they exercised the various components of the C language and often had test data.

The results of our current version of Convert2Java are encouraging. Many of the problems requiring user intervention will be corrected in the next release of the translator. The errors listed as *Library* can be corrected by writing the corresponding Java code for a missing C library function. Often, this is a simple matter of adding the two or three line function code to the library of functions to be translated.

Table 1 represents the results of running a variety of program from various data structures texts. Sizes A, B, C, and D indicate the code is 0-99 lines, 100-199 lines, 200-299 lines, and 300 lines and over. The integers indicate the number of programs which experienced each problem.

Table 1: Test Results for Assorted Test Suite

Size	Total	No	Minor Modifications																		Lib		F
	Files	Errors	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
A	25	21														3	1						1
B	12	8	1	1			1									1			1				
C	13	10																1	1	1			

The problems are classified as either minor (user fixable), libary (missing functions) or fatal and

Minor Modifications

a Java requires main to have a return value, if non void. Return inserted.

b Statement in C which translates to multiple Java statements must not be used as a condition. Make a separate statement.

c EOF or EOLN not set properly.

d C uses Java reserved word as a variable. Rename variable.

e Replace getchar with scanf. Replace putchar with printf.

f `\a` isn't supported. Remove

g Java does not allow empty loop. Restructure.

h Java requires casting that C does not require.

i C to Java compiler does not support comparison of two pointers. Restructure.

j Compiler does not permit some forms of declarations. Restructure.

k User must supply correct version of deep copy for data structure.

l Remove static designation inserted by compiler.

m Variable length argument list. Add Vector access.

n Warning message, but results identical.

o Java doesn't allow unnamed block scope variable to have same identifier name as outer scope. Message is "Variable is already defined in this method." Solution is to rename variable.

p Had to fix uninitialized array, as Java requires initialization [vH97].

q Had to remove goto statement, which is not supported.

r Argument to a function cannot be repeated without changing value (such as doit(rand())). Since arguments may be manipulated in several statements, this causes a problem. Solution it to save argument in a temporary variable in original code.

Missing Libraries:

s Random access file is not supported. Must create functions to support.

t Function not supported: ungetc, fgets, sscanf, setjmp.

Fail:

u Pointers to functions are not supported.

Figure 10: Types of Conversion Problems

are coded using the scheme of Figure 10.

Table 2 represents the results of running all the test programs from [JK95]. Note that over 90% of the programs can be translated with no modifications or with minor modifications. This means that Convert2Java is a viable tool for individuals wishing to convert large amounts of C code to Java.

8 Related Work

There are several efforts to convert from Java to C for efficiency [PTB+97]. Laffra's C2J provides a line by line translation but does not have the power to examine global context nor deal with complicated translation issues [Laf96].

References

[CW94] R. Crutchfield and D.A. Workman. Quality guidelines = designer metrics. In *Proceedings of the 1994 conference on TRI-Ada '94*, pages 29–40, January? 1994.

[GFKH99] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-Area Computing: Resource Sharing on a Large Scale. *Computing Practices*, 32(5):29–37, 1999.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 Conference on Supercomputing*, December 1995.

[JA99] N. Jensen and V.H. Allan. C to Java: A Look a the Speed Decrease. Technical Report USU-99-101, Utah State University - Computer Science Department, Logan, UT, June 1999.

[JK95] R. Johnsonbaugh and M. Kalin. *Applications Programming in ANSI C*. Prentice–Hall, Inc., Upper Saddle River, NJ 07458, 1995.

[Laf96] C. Laffra. *Advanced Java : Idioms, Pitfalls, Styles and Programming Tips*. Prentice–Hall, Inc., 1996.

[PTB+97] T.A. Proebsting, G. Townsend, P Bridges, J.H. Hartman, T. Newsham, and S.A. Watterson. Toba: Java for Applications A Way Ahead of Time (WAT) Compiler). In *Proceedings of the Third Conference on Object-OrientedTechnologies and Systems (COOTS'97)*, January 1997.

[vH97] A. van Hoff. The Case for Java as a Programming Language. *IEEE Internet Computing*, 1(1):51–56, January-February 1997.

Table 2: Test Results for Johnsonbaugh Suite

Program/Size		Minor Modifications																		Lib		F
		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
kinder	A	•																				
convert	A	•																				
taxes1	A	•																				
taxes2	A		•																			
stat	A	•		•																		
bargraph	A																					
average	A																					
chksort	A																					
primes	A																					
calendar	A	•																				
bargrph2	A			•		•																
grade	A				•																	
register	B				•																	
dice	B						•	•														
fact	A																					
tower	A																					
walk_nr	A																					
walk_r	A																					
tromino	B																					
max	A		•											•								
line_len	A																					
cars2	A																					
matmult	A					•																
phone	C																			•		
perc	B								•													
reverse	A									•												
timer	B																					•
page	A																				•	
sched	B																				•	
size	A					•																
films	B		•																			
sales	A																				•	
elephnt2	A										•											
line_ed	D																				•	
queue1	B																					
queue2	C																					
stack1	B											•										
stack2	B											•										
infix	C					•																
bfs_adj	B																					
bfs_link	C																					
astar	D												•									
assert	C								•													
jump	A																					•