1

Code Migration Through Transformations: An Experience Report

Kontogiannis, K.[1], Martin, J.[2], Wong, K.[2], Gregory, R.[3],
Mueller, H.[2] and Mylopoulos, J.[3]

University of Waterloo[1]
Dept. of Electrical Eng.

University of Victoria[2]
Dept. of Computer Science

University of Toronto[3]
Dept. of Computer Science

Abstract

One approach to dealing with spiraling maintenance costs, manpower shortages and frequent breakdowns for legacy code is to "migrate" the code into a new platform and/or programming language. The objective of this paper is to explore the feasibility of semi-automating such a migration process in the presence of performance and other constraints for the migrant code. In particular, the paper reports on an experiment involving a medium-size software system written in PL/IX. Several modules of the system were migrated to C++, first by hand and then through a semi-automatic tool. After discovering that the migrant code was performing up to 50% slower than the original, a second migration effort was conducted which improved the performance of the migrant code substantially. The paper reports on the transformation techniques used by the transformation process and the effectiveness of the prototype tools that were developed. In addition, the paper presents preliminary results on the evaluation of the experiment.

1 Introduction

Legacy software systems are software systems that have been in operation for many years, have evolved to meet changing organizational demands and computing platforms, and are often mission critical for the organization that owns and operates them. Managing such systems is difficult because of frequent breakdowns, spiraling maintenance costs and shortages of qualified personnel who are willing to work with obsolete programming languages and operating platforms. Not surprisingly, management is looking for alternatives, which sometimes take the route of totally replacing the legacy system in question with a new one, or re-engineering it.

Unfortunately, re-engineering a large system not only requires a very high commitment of human resources, but also introduces a number of risk factors such as integration errors, introduction of faulty code and non-compliance with global constraints on performance, maintainability, etc.

One pragmatic approach to the re-engineering of legacy code is to "migrate" the code into a new platform and/or programming language. Migration can be done at different levels which are increasing more ambitious and time-consuming to implement. At lowest levels, migration takes the form of transforming (or, "transliterating") the code from one language into another. At higher levels, the structure of the system may be changed as well to make it, for instance, more object-oriented. At still higher levels, the global architecture of the system may be changed as part of the migration process. For this paper, we adopt a lower level migration strategy, because it is practical, it does not require that the software re-engineer is familiar with the legacy code, and is most amenable to automation.

Legacy code migration is rarely done in a vacuum. Instead, for each migration project there are requirements such as "the migrant code must run at least as fast as the original code", or "the migrant system must be easier to maintain". These non-functional requirements introduce a risk factor into the migration process, since they can usually only be evaluated after the migration is complete. Another constraint often adopted in order to reduce the risk of migration projects is that the process must be incremental [Brodie95], i.e., can be conducted so that certain components of the legacy system are selected and migrated, resulting in an operational system. Such an

*This work was funded by IBM Canada Ltd. through the Center for Advanced Studies (Toronto), the Natural Sciences and Engineering Research Council (NSERC), and the National Research Council of Canada. The IBM contact for this paper is Bill O'Farrell billo@ca.ibm.com

incremental process ensures that migration can proceed in a piecemeal fashion, thereby lowering the risk of overall failure, cost and time overruns, and the like.

This paper reports on an experiment intended to develop a methodology for building code migration tools which meet non-functional requirements and support incremental migration. In particular, the paper reports on the migration of components of a medium-size software system from PL/IX to C++. PL/IX is a dialect of PL/I that has been used widely for system development. As part of the experiment, several modules of the system were migrated to C++, first by hand and then through a semi-automatic tool. After discovering that the migrant code was performing up to 50migration effort was conducted which improved the performance of the migrant code substantially. In both phases, migration was first carried out by human experts, followed by identification of the heuristics used by the experts, and then coding of these into the tools that semi-automate the migration process. The transformation tool has been built within the framework of the Consortium for Software Engineering Research and it took roughly 8 person months to develop.

The case study used for the experiment involves a compiler optimizer written in PL/IX and consisting of approximately 300KLOC. The "owners" of the system have variable degrees of familiarity with the global architecture of the system and extensive familiarity with particular components they are responsible for. The coding standards and the informal information (comments, variable names) are key guiding elements for the understanding and maintenance of the system. The system implements sophisticated code optimization techniques and has been highly optimized itself during its 15-year lifetime. Some components of the system are not owned by any member of the development team and are therefore very difficult to maintain. Not surprisingly, the team is reluctant to perform radical changes to its structure since this may affect negatively its overall performance. For such a case, code transformation is an attractive alternative, promising to move parts of the system, for instance, orphaned components, into a modern programming language (C++), without any deterioration in performance, nor a high risk of failure .

Section 2 of the paper reviews the literature for related research, while section 3 describes the transformation process adopted for the project. In section 4 we discuss some of the migration tactics used for data types, which in some cases (e.g., for aggregate types) are substantially different in the two languages. Sections 5 and 6 describe the the transformation process, while section 7 discusses integration and porting issues for migrated modules, also some of the limitations of the tools that have been developed. Finally, section 8 presents an overall evaluation of the experiment and section 9 offers conclusions and describes directions for further research.

2 Related Work

A number of research teams have addressed the issue of source code migration. In [Gillesp98] a tool that is used to transform Pascal programs to C , while in [Feldman93] a Fortran-77 to C and C++ converter is proposed. Both migration tools are based on parse trees that are generated from source code, are consequently transformed and then fed to a compiler of the target language. In [Yasuma95] a system for translating Smalltalk programs into C is presented which creates runtime replacement classes that implement the same functionality as Smalltalk classes in the source code. A semi-conservative, real-time garbage collection mechanism is also provided in this environment. On the commercial front, a Fortran to Fortran-Windows converter has been developed by [Sigma98] and allows for HP UX, Sun, VAX, and IBM Fortran. to be converted to Fortran that can run on Windows/98/NT environments. Similarly, a number of translators for Ada 83 to Ada 95, CMS-2 to Ada, Jovial to Ada, Fortran to Ada have been developed by Xinotech [Xino98].

In many respects, the translation problem of code from one programming language to another can be thought as a problem of as mapping syntactic and semantic patterns of the source language to patterns in the target language. Within the framework of pattern identification, Baker ([Baker94]) represents source code as a stream of strings. The approach uses parameterized pattern matching techniques based on a variation of the Boyer-Moore algorithm to identify duplication within a string. Paul ([Paul94]) proposes a system (SCRUPLE) in which regular-expressions are used to locate programming patterns in a large software system. Likewise, Johnson ([Johnson94]) identifies text-level patterns in source code by computing fingerprints using a hashing mechanism. These are then compared to identify similarities between two texts corresponding to code fragments.

Finally, other approaches, originally developed for the area of syntax based editing, include CIA

([Chen90]), CSCOPE ([Steffen85]), the Pan system ([Ballance92]), the CENTAUR system ([Borras88]), the Cornell Synthesizer Generator ([Reps84]) and, A* ([Ladd95]).

3 The Transformation Process

In order to perform software transformations, it is useful to represent source code at a higher level of abstraction than, say, text or even parse trees [Yasuma95]. Like many other research efforts, we have chosen to represent the source code in the form of Abstract Syntax Trees (AST) which has been emitted from a custom-built PL/IX parser and linker. Abstract Syntax Trees provide a fine-granularity representation of source code details (type information, scoping, resource usage) which is exploited throughout the transformation process. A custom-built PL/IX language domain model provides an hierarchical view of the PL/IX language constructs and "drives" the set of transformation routines which maps PL/IX language constructs to corresponding C and C++ constructs. Once a PL/IX construct is identified, a transformation routine is selected and invoked. Each entity of the custom-made PL/IX domain model has a corresponding transformation routine. For example a `StatementIf` in PL/IX will be transformed to an equivalent statement in C by transforming its `condition`, its `then-clause`, and its `else-clause`, which the domain model indicates that are of type `Expression`, `Statement`, and `Statement` respectively (see example in Section 6).

A subset of the PL/IX domain model is illustrated below, with indentation representing sub-classing:

```
PlixObject
       PlixApplication
       PlixFile
       :  PlixProgramFile
       :  PlixIncludeFile
       Statement
       :  StatementIf
       :  StatementAssignment
       :  StatementCall
       :  StatementDeclare
       :  :   DeclareObject
       :  :   StatementInclude
       :  StatementDoEnd
       Expression
       :  ArithmeticAdd
       :  ArithmeticDivide
       :  ArithmeticMultiply
       :  ArithmeticSubtract
       :  IdentifierReference
       :  LogicalAnd
       :  LogicalNot
       :  LogicalOr
       :  ReferenceComponent
```

Each language construct is represented in the domain model as an object with a list of attributes. For example a PL/IX `If Statement` is represented as:

```
StatementIf object
  Subclass-of Statement
  Attributes
     condition   : Expression
     then-clause : Statement
     else-clause : Statement
End StatementIf
```

The transformation of a given code fragment consists of the following steps:

Transformation of its data structures: PL/IX data structures of the subject system were analyzed and migrated based on a set of transformation rules applied both to the basic and to aggregate data types of the source language. Overlays, unnamed fields, memory areas, aliases, controlled variables, and many other complex features of the source language were mapped to functionally similar structures in the target language.

Generation of supporting utilities and other target language specific artifacts: In many cases constructs of the source language do not have obvious corresponding constructs in the target language. In this case, a set of utilities and libraries need be constructed to support execution of the migrant code. Examples of such constructs include array classes (one dimensional or multidimensional), memory allocation and deallocation mechanisms, and built-in functions that are specific to the source language.

Generation of the new system: Programs are mapped from the source language to the target language in a way that preserves structure, algorithmic patterns, informal information (i.e variable names), comments, and indentation. In particular comments are handled as annotations in the AST and are extracted by the custom-made linker. In most cases map very well to the correct position in the new generated code as most comments occupy a single line and do not interleave with expressions inside a source code statement. For cases where there is a position discrepancy

this usually is in the order of one or two lines and can be easily identified at the code inspection phase.

The process of selecting which components of the legacy system to migrate begins with program understanding and re-documentation techniques which lead to a detailed analysis of the legacy system in terms of its major components, files, subsystems, and corresponding interfaces [Finni97]. Each subsystem is thoroughly analyzed in terms of its control and data flow properties, its interface with the rest of the system, and the major algorithms and data structures used.

Once a global decomposition of the legacy system has been achieved in terms of subsystems and interfaces, a number of subsystems can be selected as the primary candidates for migration. The selection criteria focus on:

Code size: Subsystems between 5-10KLOC are ideal candidates since they can be manually examined and inspected.

Interface patterns: Self contained, systems with simple interface patterns, and as few as possible side effects, are also primary candidates.

Maintainability: Subsystems which degrade in quality and performance due to the lack of resources are also good candidates for migration. Maintenance for these systems is performed on as needed basis and usually is of a corrective variety. The incorporation of substantial domain knowledge in the code (i.e. domain specific algorithms) results in increased difficulty on understanding and maintaining the particular subsystem.

Performance: Subsystems that have been identified as a performance bottleneck - time or space - during testing due to, algorithmic complexity or poor utilization of language constructs are also high on the candidate list.

The following subsections discuss in detail the issues involved for the migration of any given subsystem.

4 Data Type Migration

The first step towards code migration is the transformation of data structures.

Basic Type Conversions: A global table that maps PL/IX primitive data types to their corresponding C/C++ data types was constructed. The type mappings are illustrated in Table.1.

Aggregate Type Conversions: Mappings for aggregate data types were developed by the code transformation program, using information on basic data type conversion. As a general rule, PL/IX records were transformed to C/C++ structures (`struct`). Overlays were handled as C/C++ unions. The type conversion table can be modified with ease during the migration process and can be tailored to particular requirements for a specific migration project. A detailed discussion on the issues involved for data type transformations can be found in [CserRep]. An example that illustrates the transformation of an aggregate data type is shown below:

```
dcl 1 dsstate external,
     2 prg integer,
     2 externs integer,
     2 statmem integer,
     2 curr_pdx_statmem integer,
     2 aregcnt integer,
     2 aregused integer,
     2 flags bit(32),
     .2 *,
       3 dbg bit,
       3 detail bit,
       3 warn bit,
       3 detail2 bit,
     2 memused bit(32),
     2 maxmuse bit(32);
```

The data structure above is automatically transformed into:

```
struct any_DSSTATE {
  int             prg;
  int             externs;
  int             statmem;
  int             curr_pdx_statmem;
  int             aregcnt;
  int             aregused;
  union FLAGS {
    int             flags;
    struct any_DSSTATE_8 {
      int             dbg:1;
      int             detail:1;
      int             warn:1;
      int             detail2:1;
    };
  };
  int             memused;
  int             maxmuse;
};
```

PL/IX Type	C Type
integer	int
integer half	short int
integer long	long int
float	double
float short	float
float long	double
character	char
bool	bool
bit	bool
bit(1)	: 1
bit(2)	: 2
bit(8)	unsigned char
bit(9)	: 9
bit(16)	unsigned short
bit(17)	: 17
bit(32)	unsigned

Table 1: Basic PL/IX to C type conversion.

Macros to access fields:

In order to replicate the access of a field that is deeply nested within a record, a macro is generated for every field access pattern found in the code. These macros allow the migrant C++ code to closely resemble its PL/IX ancestor. The syntax and the definitions for calling these macros are stored in tables during a preprocessing phase. Another transformation issue relates to PL/IX array references for which indexes do not necessarily appear next to the element in the reference component to which they belong. For example, in field access `a.b(i)`, it may be the case that `a` is an array, not `b` as it may be thought initially. An elaborate analysis of the PL/IX data structures, and the generated corresponding C++ data structures is required in order to construct the correct transformation for such a field access. This analysis automatically reconstructs the full path of the corresponding C++ structure, starting from the last, inner-most, field on the reference component and proceeding towards the outer-most field of the structure. Array indices are substituted, from left to right for each array related field found in the reconstructed path. "Liked" fields are also handled appropriately.

Ambiguous field names: The problem here is that PL/IX allows paths of field accesses to be abbreviated (for instance, *a.b.c* is abbreviated as *c*). Such paths need to be reconstructed for the migrant C++ code. The first step in dealing with this problem is finding field names that are used more than once within the same compiler option. A dis-ambiguation process then locates all such names and builds a path of fields that is required to access each such field. Once those paths are available, a subset of those chains is selected which completely distinguishes among the ambiguous fields. For example, a record rec_1 may consist of field $field_a$ which consists of nested fields $field_b$ and $field_c$. One can then write an assignment statement in PL/IX as $field_a$ `= val` (as opposed to $field_a.field_b.field_c$ `= x`). The migration tool builds a macro for each previously ambiguous identifier and this macro definition is consequently stored in a global macro table. Examples of macro definitions for transforming reference components are illustrated below:

Uses of field access for the `bb_begin`, `bb_end`, `bb_count`, and `irreducible` PL/IX fields (sample from the original PP/IX code):

```
do plx= bc(bb_end(bb))
   repeat(bc(plx))
   while(plx^=bb_begin(bb));
      Statements
        -------------
do bb= 2 to bb_count;
      Statements
        -------------
irred= irreducible(rgn);
```

the C generated code for the PL/IX code segments above is:

```
for (plx = bc[C_BB_END(bb)];
     plx != C_BB_BEGIN(bb);
     plx = bc[plx]) {
         Statements
        -------------
for (bb = 2; bb <= C_BB_COUNT; bb++) {
         Statements
        -------------
irred = C_IRREDUCIBLE(rgn);
```

while the C corresponding macro definitions for the field accesses are:

```
#define C_BB_BEGIN(i)
     (bb_tab.bb_t[(i)].BB_BEGIN.bb_begin)
```

```
#define C_BB_END(i)
     (bb_tab.bb_t[(i)].bb_end)
#define C_BB_COUNT
     (bb_tab.bb_count)
#define C_IRREDUCIBLE(i)
     (bb_tab.bb_t[(i)].flags.irreducible)
```

Literals: In this phase of the migration process, we look for all PL/IX literal declarations and attempt to classify them into one of six types. These types are as follows:

1. The literal is an integer.
2. The literal is an ambiguation
3. The literal is a string of PL/IX variable attributes
4. The literal is a string that is an Identifier
5. The literal is a string that is a PL/IX type
6. The literal is a bit mask

All of these cases are handled either by declaring appropriate macros evaluating on the correct value of the literal definition, or by using the C preprocessor (i.e. *#define*) to define literal constant values. Examples of literal definitions include:

The PL/IX code for literal definitions is:

```
dcl mode32  integer constant init(0);
dcl tags_full  lit('3');
dcl tags_small  lit('2');
```

and the corresponding generated C code for the PL/IX literal definitions above is:

```
const int mode32   = 0;
#define C_TAGS_FULL  3;
#define C_TAGS_SMALL 2;
```

5 Generating Class Header Files

The second step for transforming a PL/IX application to C/C++ was to generate the C/C++ header files using the data type information obtained from the PL/IX Abstract Syntax Tree, the data type tables, and the macro definition tables. Generating the header files required the following steps:

Constructing the Foundation Classes: This first step deals with the construction of a set of "foundation" classes used in the header files. Each of these classes is defined to have only static methods in order to resemble the PL/IX semantics. For every public procedure in PL/IX, one class with the procedure as its public method is created.

A PL/IX procedure group is transformed into a class with multiple public methods. Every nested procedure, or unexposed procedure in a group, is made into a private method of the class which holds its outer procedure (or group). This is not exactly orthodox object-oriented programming (O-O-OP), but it allows us to encapsulate a PL/IX procedure within a C++ class so that new functionality can be added to the procedure (for example, a nested PL/IX procedure) in a transparent with respect to the rest of the legacy system way. An example of this type of transformation is illustrated below:

Given a the PL/IX procedure declaration, which defines a nested procedure:

```
dslvbb: proc(bb);
  ..........
  proc_ensc: proc(memref);
  ..........
  end proc_ensc
end dslvbb
```

the corresponding C++ foundation class are generated as follows:

```
class Dslvbb {
public:
  static void  dslvbb(int bb);
private:
  static void proc_ensc(any_L_BAG memref);
};
```

Merging Classes: The next step in the process of creating class header files is merging classes that correspond to identical types for parameters passed in several procedures. For example, if we have procedures P1 and P2 with parameters p1 of type T1, and p2 of type T2, then we want to map P1 and P2 into methods of the class C_T1 corresponding to the type T1. A Java based interface has been built for to make this process interactive and allow the user to specify how the classes corresponding to procedures are to be

merged. As the reader might infer, a data type that is heavily used as a parameter to several procedures is a primary candidate to be transformed to a class and the procedures to be member functions with the particular parameter removed. As an example consider the `scanner` PL/IX data type which is transformed to a class as follows:

```
class Scanner {
private:
  static      alloc(int lim);
  static      new_item(int elem, P_BAG flags);
  void        join(int lo, int hi,
                   boolean ds_used_bit);
public:
  void        init_scan(int lst);
  int         advance();
  int         pick();
  boolean     ds_is_used();
  static int  make_empty_list();
  static      void kill_list(int list);
  int         find_item(int elem);
  void        delete_item();
  void        delete_lfrag();
  void        insert_mem_dsc(Scanner sc,
                   boolean ds_used_bit);
  boolean     join_item(int m,
                   boolean ds_used_bit);
  void        insert_live(int item,
                   P_BAG attrs);
  void        insert_after(int elem,
                   P_BAG p_flags);
  void        insert_item(int elem,
                   P_BAG p_flags);
  boolean     join_live(Scanner join_me);
  void        split_item(int m);
  void        constrict_item(int m);
  boolean     defrag_live(boolean ds_u);
};
```

Handling External Variables: After processing all of the declarations in the original PL/IX code, all of the external variables are stored in a global table. These declarations are printed to a file that is not included anywhere in the new C++ code, but references to the variables in it are made via *extern* statements. An alternative solution is to include this file in all others and remove the corresponding *extern* declarations.

6 Source Code Transformations

The main objective of this phase is to write out C/C++ source code that can be compiled with minimal corrections. A set of transformation routines modeled after the custom-built PL/IX domain model (referred to as the "skeleton") is the driver for all transformations in the system. A tree traversal routine, traverses a PL/IX Abstract Syntax Tree and for each AST node corresponding to a PL/IX element that needs to be transformed, a transformation and formatting function is invoked. For example when transforming the body of a procedure the routine that handles the transformation of `Statement` is invoked. A selection process identifies what type of `Statement` is to be transformed and the appropriate transformation routine is invoked.

An example on how the transformation process is applied to the PL/IX Abstract Syntax Tree that corresponds to an `StatementIf` is shown below. The transformation routine proceeds as follows:

```
PL/IX Code:

  if ^kill then cannot_kill = true;


Corresponding AST Node:

  #1445 <a statementif>
  class:  STATEMENTIF
  parent-expr: #1456<a statementdoassigment>
  condition: #1457<a logicalnot>
  then-clause: #1458<a statementassignment>


Corresponding Domain Model:

  StatementIf
    condition  : Expression
    then-clause: Statement


Transformation Process Trace:

  Transform Statement
  .. Transform StatementIf
  -->  "if ("
  .... Transform Expression
  ...... Transform LogicalNot
  -->     "(!"
  ........ Transform IdentifierReference
  -->        "kill"
  -->      ")"
  -->   ")"
  .... Transform Statement
  ...... Transform StatementAssignment
  -->   ...................
  -->   ...................
  -->   "cannot_kill = true;
```

```
Code Translates To:

  if ( (!kill) ) {
    cannot_kill = true; }
```

Formatting for nested functions is generated through the use of a two way pass. A first pass on the AST is used to collect all necessary information, while second pass generates the migrant code. Note that in order to conform with PL/IX semantics, nested procedures have been defined as private methods on the "basic" class encapsulating the top level procedure (see data type transformations).

Syntactically correct C++ code is output to a `.h` or to a `.cpp` file. As expected, this process does not always guarantee that the resulting code will be semantically correct, with respect to the original application code. However, the process provides a fast and relatively reliable way to handle massive volumes of code and thus makes code migration a feasible alternative. The final result of this step is checked manually, and the generated code is passed through *all* test buckets to verify the correctness of the migrant code.

A number of helper libraries have also been defined for built-in PL/IX functions, such as those for memory allocation. For example, an `ArrayALLOCATION(x)` function is defined as a macro and corresponds to the PL/IX statement `allocation(x)`. Helper data types are also used to simplify the appearance of the generated code, and to make it look as much as possible to the original PL/IX code [1]. These types inherit from basic library elements such as `set`, `bag`, `list`, `sequence` and `array`. For example a template class PlixArray< *type* > was used to model one dimensional arrays.

7 Integration, Porting, Limitations

Once one or more subsystems of the legacy system have been migrated, they need to be integrated with other existing PL/IX subsystems. This integration requires (i) writing of emulation classes for PL/IX built-in data types and functions, (ii) manual fixing of the generated code where necessary, and (iii) writing of interface code, so the C++ code can access PL/IX functions and data types. The last two steps can actually be carried out concurrently, by compiling the modules file-by-file, and then adding interfaces for routines and functions where necessary. Since the modules share most of the interface code, less interface code needs to be written for each new module so that the last step requires minimal effort after the migration of several modules. We have identified a number of manual changes required for the new code to compile and link properly. These relate to handling multi-dimensional arrays, and field accesses and to limitations on mapping the PL/IX semantics to C/C++ semantics (i.e. in the case of LEAVE statement in PL/IX which has to be mapped to some form of GOTO in C/C++). Some of these manual changes have either been incorporated in the tool or been flagged so that the developers can identify easily the points that manual intervention is required.

The migration process has limitations, including:

- *Local variables of nested procedures*: C++ does not have nested procedures, so local variables that are used in sub-procedures, need to be passed as parameters to the sub-procedures (or be made class variables)
- *Insertion of gotos and labels*: multi-level leave/iterate statements have to be done automatically
- *Grouping sub-procedures*: static procedures of a class will not work with the current treatment for recursive procedures. A design has been proposed on how to fix the problem and will be implemented with the next release of the transformation tool.

8 Tool Evaluation

Evaluation of the migration tool focused on four major areas. The first area examines the generality of the tool in terms of the lessons learned on how the same transformation process can adopted in order to migrate and transform applications written in similar to PL/IX languages (i.e. PL/I). The second area, examines the performance of the new compiler with respect to memory usage, the third investigates the performance of the migrant code compared to the original, while the fourth examines reductions in the human effort required to migrate legacy code.

The main characteristics of the transformation process as compared to other transformation tools

[1] This was another requirements for the migration effort, in order to facilitate future maintenance of the migrant code

Subsystem	PL/IX Time (*in minutes*)	C++ Time (*in minutes*)	# of Runs
Test13k	11.96	12.19	956
gcc	70.52	70.30	121
lib g++	0.916	0.8626	1065
JPEG	9.4257	9.106	764

Table 2: Compilation time statistics with the old (PL/IX) and the new (C++) compiler. Results were averaged on the number of runs. The experiments were conducted on four different models of RISC/6000 machines.

Subsystem	Code Size (KLOC)	Transformation Time (*min:sec*)
BI	6.2KLOC	18:01 (9:04)
DS	7.8KLOC	28:12 (13:10)
SM	7.4KLOC	26:45 (14:00)

Table 3: The time to transform the selected subsystems after parsing. Code size refers to actual code excluding comments and blank lines.

[Verhoef97] is the use of a repository that holds all the necessary information for the system to be transformed, as opposed to applying transformations at the syntactic/semantic level using grammar and semantic actions re-write rules. This repository is populated at parse time and is annotated appropriately by a custom made for PL/IX linker. All code transformations have been built in terms of modular application programs at the repository level. It has become apparent from this project, that the transformation logic should be as modular and localized as possible (i.e. different transformation programs for each language construct), and focus on three main points; *a)* transformation of data types, *b)* development of support utilities (i.e. macros, interface classes) and, *c)* transformation of the source code entities (i.e. the actual application). The schema in the repository has been built in such a way as to be compatible with the PL/AS and PL/X domain model. Therefore, the transformation process is also applicable, with minor schema and transformation logic changes, to applications written in PL/AS and PL/X. We currently investigate the possibility of applying the same transformation logic to applications written in PL/I.

As indicated in the introduction, the migration tool was built in two phases. The first phase aimed at developing a tool which would produce C and C++ code that resembled substantially the hand-transformed code produced for a legacy subsystem. The second phase aimed on incorporating into the transformation routines heuristics that have been used to hand-optimize the migrant code. The migration tool has been tested with three subsystems of the legacy system. The performance results obtained for these subsystems are reported in Table 2.

As indicated earlier, the legacy system that has been partly migrated is a compiler optimizer. In order to evaluate the performance of the new version of the optimizer (which includes the three migrated subsystems), the optimizer was run on four sets of source code, namely:

i) A single 13,000 line pre-processed C source file, about 7,000 lines of which were actual code (rather than declarations); *ii)* The Independent JPEG Group's free JPEG software, sixth public release (C sources); *iii)* Stage 1 of the GNU C compiler, version 2.7.2.2 (C sources) and; *iv)* The GNU C++ library, version 2.7.1 (C++ sources)

The performance results obtained when running the optimizer against these four test cases with the PL/IX and C++ versions of the subsystems respectively are reported in Table 2. These are preliminary results obtained after the optimization heuristics were added in the transformation tool. More results are being collected as the migration tool is undergoing further enhancement related to heuristics that can be incorporated to the transformation logic and used for enhancing the performance of the new code.

The time it takes the tool to generate migrant code as a function of the size of the input code (time for parsing is shown in parentheses) as well as the effort in person days for adapting and integrating the migrated code with the rest of the original legacy system are illustrated in Tables 3 and 4 respectively.

These results indicate a significant enhancement in productivity for the migration process, with no deterioration in the performance of the migrated code. In particular, manual transformation of one subsystem (DS) required approximately 50 person-days. By comparison, it took only one person day to adapt the automatically generated migrant code for the same subsystem so that it can be compiled, linked, and integrated with the rest of the legacy system. Statistics gathered for the other two subsystems (BI and SM) follow the

Subsystem	Code Size (KLOC)	Automatic Transliteration (*person days*)
BI	6.2KLOC	0.3pd
SM	7.4KLOC	2pd
DS	7.8KLOC	1pd
Subsystem	Code Size	Manual Transliteration
DS	6.2KLOC	50pd
Subsystem	Code Size	Manual Optimization
DS	6.2KLOC	10pd

Table 4: The effort to adapt and integrate the new components to the rest of the system compared to the effort to manually transform the same subsystem.

Subsystem	Subsystem Size (KLOC)	Original System Size (MB)	New System Size (MB)
BI	6.2KLOC	27.150	27.265
DS	7.8KLOC	26.738	27.976
SM	7.4KLOC	26.687	28.172

Table 5: The total size of the compiled system (binary) before and after migration.

same trend and indicate an effort of approximately half a person-day and two person-days respectively. The difficulty of the integration task depends largely on the usage patterns of the data structures in the legacy system which have not been migrated yet and therefore need be shadowed, so that the new version of the system can be compiled and linked as a whole. Finally, in Table 5 the size of the old and the new version of the overall system is compared. As it is shown, the new system is approximately 5% larger in size than the original one. This can be explained by the added C++ libraries to handle and simulate the behavior of several PL/IX language-specific constructs.

The comparison of the PL/IX to C converter discussed in this paper, with the other tools found in the literature is summarized in the following points. In [Yasuma95] the authors present a higher performance ratio (that is the performance of the old system divided by the performance of the new system) for the migrant code (ratio = 1.22) than the performance ratio we could obtain on our initial experiments (ratio = 1.02). The reason for this discrepancy is that we deal both with PL/IX, which is a highly optimized language (as opposed to SmallTalk), and with a highly optimized application which, in our case, is the IBM compiler back-end. Therefore our margins of performance improvement are limited compared to the ones that could be obtained in SmallTalk to C conversion.

In [Feldman93] and [Gillesp98] the authors report that the performance of the converted code using the `f2c` and `p2c` utilities respectively, is the same as the performance of the original code, indicating thus a performance ratio of 1.0. However, the code generated by these tools is reported to be non-maintainable due to the structure and the characteristics of the generated code (no informal information and code structure preserved). This is to be expected since these tools were meant to produce code, which after compilation is binary-equivalent with respect to the original code. We took a different approach, and we generate code that is structurally "similar" with respect to the original code so that we could still produce portable and maintainable code without of course affecting the performance.

On these aspects, our tool performs well, as it both maintains an acceptable performance ratio, and allows for the new code to be more maintainable and portable due to resources, analysis tools, and the range of existing compilers available for C-based systems, as compared to limited support that is available for PL/IX-based systems. In this respect, we can say that our tool could be placed in the middle of the spectrum represented in its extremes by: *i)* the SmallTalk to C converter which deals with low optimized SmallTalk constructs and not compatible programming paradigms between the source and the target language and on the other end by *ii)* the Fortran to C converter which deals with easier mappings between the source and the target language, but with no further maintainability and portability requirements for the target system.

9 Conclusion

We have described an experiment in code migration in the presence of global constraints, namely non-deterioration in the performance of the maintainable migrant code, and an incremental migration process. Our experiment suggests that it is possible to develop tools which reduce the human effort required for the migration process by one to two orders of magnitude, while respecting such constraints.

We are currently completing a thorough evaluation of the migrant code in comparison to its legacy ancestor. We are also in the process of enhancing our

migration tools to meet maintainability and other non-functional requirements for the migrant code.

Acknowlgement

The authors would like to thank Bill O'Farrell and Stephen Perelgut of IBM Center for Advanced Studies, for their technical and management support without which this effort would not be possible. We would also like to thank Greg Mori for his efforts on the first phase of the tool implementation.

About the authors

Kostas Kontogiannis is an Assistant Professor at the Electrical & Computer Engineering department, University of Waterloo. His interests include software re-engineering, software migration and, distributed object technology. Johannes Martin is a Ph.D candidate at the Computer Science department, University of Victoria. His research interests include software architectures, software re-engineering and, tools for source code visualization. Kenny Wong is a post-doctoral Research Associate at the department of Computer Science, University of Victoria. His interests include software migration, tools for visualization of software architectures, and distributed systems. Richard Gregory is a fourth year student at the Computer Science department, University of Toronto. His interests include software re-engineering and network-centric computing. Hausi Muller is a Professor at the University of Victoria, department of Computer Science. His interests include software visualization, distributed object technology and network-centric computing. John Mylopoulos is a Professor at the University of Toronto, department of Computer Science. His interests include software requirements, conceptual modeling, software repositories, and software re-engineering.

References

[Baker94] Baker S. B, "Parameterized Pattern Matching: Algorithms and Applications", *Journal Computer and System Sciences*, 1994.

[Ballance92] Ballance, R., Graham, S., Van De Vanter, M., "The Pan Language-Based Editing System", *ACM Transactions on Software Engineering and Methodology*, Jan. 1992, Vol. 1, No.1, pp.95-127.

[Borras88] Borras, P., Clement, D., Despeyroux, Th., Khan , J., Lang, G., Pasual, V., "CENTAUR: The System", *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston, Mass, 1988.

[Brodie95] Brodie, M., Stonebraker, M., "Migrating Legacy Systems", Morgan Kaufman Publishers, 1995.

[Chen90] Chen, Y., Nishimoto, M., Ramamoorthy, C., "The C Information Abstraction System.", *IEEE Transactions on Software Engineering*, vol.16, No.3, 1990, pp.32 5-334.

[CserRep] Consortium for Software Engineering Research: http://www.cser.ca

[Feldman93] Feldman, S., Gay, D., Maimone, M., Schryer, N., "A Fortran to C Converter", AT&T Technical Report No. 149, 1993.

[Finni97] Finnigan , P. et.al "The Software Bookshelf", *IBM Systems Journal*, vol.36, No.4, 1997.

[Gillesp98] Gillespie, D., "A Pascal To C Converter", *The HP-UX Porting and Archive Center*, http://hpux.u-aizu.ac.jp/hppd/hpux/Languages/p2c-1.20/readme.html

[Johnson94] Johnson, H., "Substring Matching for Clon e Detection and Change Tracking", *International Conference on Software Maintenance* 1994, Victoria BC, 21-23 September, 1994, pp.120-126.

[Ladd95] Ladd, D., Ramming, J., "A^*: a Language for Implementing Language Processors", *IEEE Transactions on Software Engineering*, vol.21, no.11, Nov. 1995, pp.894-901.

[Paul94] Paul, S., Prakash, A., "A Framework for Source Code Search Using Program Patterns", *IEEE Transactions on Software Engineering*, June 1994, Vol. 20, No.6, pp. 463-475.

[Reps84] Reps, T., Teitelbaum, T., "The Synthesizer Generator", *In Proc. of the SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Pittsburgh PA, 1984, pp.42-48.

[Sigma98] Sigma Research Inc. : http://www.sigma-research.com/for2win

[Steffen85] Steffen, J., "Interactive examination of a C program with Csope", *Proceedings USENIX Assoc.*, Winter Conference, Jan. 1985.

[Yasuma95] Yasumatsu, K., Doi, N., "Spice: A System for Translating SmallTalk Programs Into a C Environment" *IEEE Transactions on Software Engineering*, vol. 21. no.11, November 1995.

[Verhoef97] van den Brand, M., Sellink, A., Verhoef, C., "Generation of Components for Software Renovation Factories from Context-free Grammars", *Proceedings Working Conference on Reverse Engineering, WCRE'97*, Amsterdam, The Netherlands, October 1997.

[Xino98] Xinotech Inc. http://www.xinotech.com