Goto Elimination Strategies in the Migration of Legacy Code to Java

Mariano Ceccato, Paolo Tonella, Cristina Matteotti
Fondazione Bruno Kessler—IRST, Trento, Italy
{ceccato, tonella, matteotti}@fbk.eu

Abstract

Legacy systems are often large and difficult to maintain, but rewriting them from scratch is usually not a viable option. Reenginering remains the only way to modernize them. We have been recently involved in a migration project aiming at porting an old, large (8 MLOC) legacy banking system to a modern architecture. The goal of the project is: (1) moving from an old, proprietary language to Java; (2) replacing ISAM indexed files with a relational database; (3) upgrading the character oriented interface to a modern GUI.

One of the steps in the migration process deals with the elimination of unstructured code (unconditional jumps such as GOTO statements). In this paper we present four alternative strategies for GOTO elimination that we evaluated in the project. Each has pros and cons, but when used in a real case, it turned out that one produced completely unreadable code, hence it was discarded. The final choice was a combination of the three remaining strategies.

1 Introduction

When a legacy system has to be ported to more recent technology, it may require to be re-designed from scratch in order to fit the major differences between past and present facilities. However, a complete re-implementation is often too expensive to carry out, because of the size of the system and the difficulty of recovering the business rules it implements. In fact, such systems are usually not well documented and the source code is often the only reliable repository of the business rules that drive the system. Considering these constraints, semi-automated migration represents often a more appropriate solution.

We have been recently involved in the migration of a legacy banking system. It consists of a large code base (8 millions lines of code) written in a proprietary language and running on character oriented displays. The system supports all the bank activities, including front office, business intelligence, account and check management. It also interacts with all the office devices, such as printers and check readers, and it produces periodic reports for various institutions, such as the central bank.

The migration involves several challenging problems, that can be grouped into three main categories:

- Changing the code from the proprietary language to Java;
- Transforming data and storage model, from ISAM indexed files to a relational database;
- Moving the character-oriented user interface to a graphical user interface.

The present paper describes our experience in addressing part of the first problem, i.e., removing the unstructured code (*goto* statements). In fact, the target language (Java) does not support them.

In this task, the objective is not limited to goto elimination, since the resulting transformed code (without gotos) is expected to be maintained (once migrated to Java) by the same developers that currently maintain the legacy system. For this reason, the final code should be as similar as possible to the original one, so the developers can take advantage of their familiarity with the old system to work on the new one. Hence, the task has indeed two conflicting objectives: (1) eliminating goto statements; (2) producing maintainable code. As a third dimension we considered also (3) the transformation speed.

Several approaches to goto elimination exist in the literature [2, 3, 7, 8, 11, 12, 16, 18]. However, existing solutions for the elimination of unstructured statements are known to have a major impact on the code quality, especially in the presence of irreducible control flow [7, 12]. Since our objective is also to avoid a deterioration of the code quality, we cannot just take any of the existing approaches and use it. We need to carefully assess the impact of each alternative approach on the code quality and then select the best compromise.

We considered four alternative approaches to goto elimination in terms of their effect on the code quality. We will discuss advantages and disadvantages of each of them and

we will motivate our final solution, which is actually a combination of three out of the four evaluated goto elimination methods. Finally, we describe what we learned from this part of the migration project, trying to distill some reusable lessons.

In Section 2 we give a description of the four goto-elimination strategies. The performance of each different strategy is presented in Section 3 and the chosen solution is described in Section 4. Related works (Section 5) and conclusions (Section 6) close the paper.

2 Strategies

Legacy systems are written in legacy programming languages, that often lack support for basic iterative constructs such as `while`, `do-while`, `for`, `break` and `continue`. On the other hand, they often support arbitrary control flow, by means of the `goto` construct. Software developers are used to write unstructured code to obtain the semantics of the missing constructs.

The banking system we are migrating to Java was developed in a BASIC-like language that originally did not support any loop construct except the for-loop. All other iterations were obtained by means of goto statements. Later on, iterative statements such as `while`, `do-while`, `break` and `continue` have been introduced in the language, but many portions of the system were already developed. Moreover, programmers were used to take advantage of gotos in several ways and they did not change radically their habits when gotos were no longer needed. Extensive usage of copy-and-paste to implement new functionalities augmented instead of reducing the number of gotos in the code. Today, among the 8 MLOC in the code base, 0.5 MLOC are goto statements.

One typical usage of gotos, that is still largely present, supports management of the user interaction state. Each code portion associated with a particular interaction state starts with a label and ends with a sequence of conditional gotos. According to the user's input and to the error conditions that may arise, the program counter is moved to another code block, representing another state of the interaction. In such code organization, it is also quite common to see jumps inside a block (i.e., in the middle of it) and not at the beginning, the reason being that according to some other part of the interaction state (usually recorded in one or more variables) part of the computation in a block must be skipped. Jumps from the middle of a block are also present. All this gives rise to an abundance of irreducible gotos, that are intrinsically hard to deal with, and to spaghetti code that is very hard to read for anybody outside the development team. Surprisingly, developers are perfectly at ease with such code organization and seem to understand it with little effort.

In this section we present the four goto-elimination strategies that were evaluated in the migration project.

2.1 Pattern based

In this approach, gotos are eliminated by identifying recurring code patterns that involve gotos and for which a straightforward and natural translation without gotos exists.

When the *do-while* statement was not available, its semantics was obtained by using *gotos*, as shown in Figure 1. If a given condition holds, either a segment of code (`stmt_seq_0`) is re-executed (`goto L0`) or it is not, and the computation goes on (`goto L1`). This corresponds to the semantics of the *do-while* statement.

```
L0:
    <stmt_seq_0>
    if (cond)
        goto L1;
    goto L0;
L1:
```

(Original)

```
L0:
    do {
        <stmt_seq_0>
    }
    while (not(cond));
L1:
```

(Transformed)

Figure 1. Pattern of goto used as do-while

Unstructured code developed in this way represents an instance of *goto* usage, that can be easily identified and transformed into equivalent, goto-less code. Moreover, the resulting code keeps a comparable degree of understandability and maintainability. Once shown to the developers, they were comfortable with it and with the automated transformation of the original code. We conclude that this transformation preserves code familiarity in a satisfactory way.

Another similar pattern is shown in Figure 2. In the *while* loop, the last part of the loop body (`stmt_seq_1`) is conditionally executed. If a given condition holds (i.e., `cond_2`), the last part is skipped and the next iteration starts. This conditional jump can be very naturally transformed into a *continue* statement.

```
L0:
    while (cond_1){
        <stmt_seq_0>
        if (cond_2)
            goto L0;
        <stmt_seq_1>
    }
```

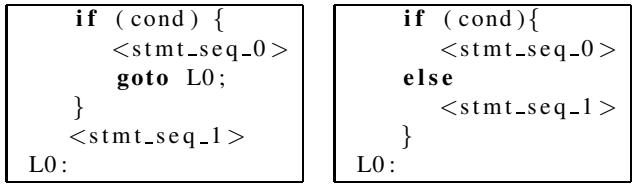
(Original)

```
L0:
    while (cond_1){
        <stmt_seq_0>
        if (cond_2)
            continue;
        <stmt_seq_1>
    }
```

(Transformed)

Figure 2. Pattern of goto used as continue

In other cases, available structured statements could be used instead of gotos. For instance, in Figure 3 a branch appears at the end of a portion of code (`stmt_seq_0`) that is executed under a given condition (`cond`). In this way, the successive portion of code (`stmt_seq_1`) is executed only if the condition does not hold. This second sequence can be moved in the *else* branch on the *if* statement.

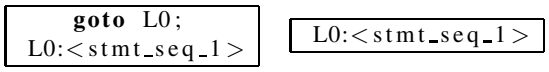


(Original) (Transformed)

Figure 3. Pattern of goto used as then

By manually inspecting a significant portion of the system, a list of patterns has been identified with the corresponding transformations. The recovered patterns are:

useless_goto : a (potentially conditional) branch to the next statement can be trivially removed (Figure 4).



(Original) (Transformed)

Figure 4. Pattern of useless goto

if_goto : a conditional branch is used to skip a portion of code (Figure 7). The goto is removed and the code to skip is moved in the *else* branch of the condition and the *then* branch remains empty. Optionally the condition can be negated and the *then* and *else* branches inverted, so as to avoid and empty *then*.

if_goto_then : a variant of the previous transformation, when the original *then* branch contains not only the *goto* statement but also some other instructions. This pattern is shown in the example in Figure 3.

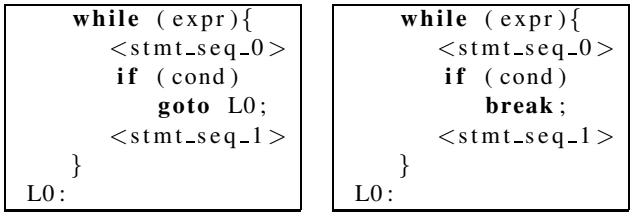
goto_within_switch : as in Figure 8, inside a *switch-case* statement a conditional branch is used to skip the execution of the rest of the current *case* and to jump outside the *switch*. The skipped code is moved into the *else* branch of the *if* and the *goto* statement is removed. No other changes are required, since the *switch* statement in our legacy language does not support case fall-through.

ifelse_goto : this pattern is shown in Figure 9. Depending on the condition of an *if* statement, either a goto (*then* branch) or a sequence of instructions (`<stmt_seq_0>` in the *else* branch) is executed. The *goto* is used only to skip statements in `<stmt_seq_0>`. Both the sequences in the *else* branch and the sequence that follows the *if* statement are executed only if the condition does not hold.

So, this pattern can be changed by inverting the condition in the *if* statement and by moving both of the sequences in its *then* branch.

continue_goto : in the body of a *while* or a *do-while* statement a *goto* is used to skip the rest of the current cycle and to go on with the next iteration. The *goto* statement can be changed into a *continue* statement (Figure 2).

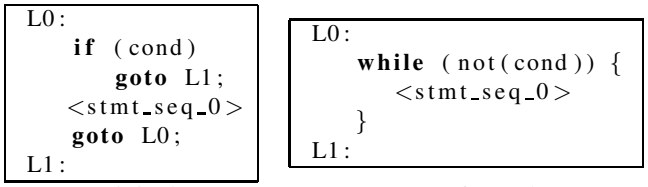
break_goto : this is a variant of the previous pattern, shown in Figure 5. A conditional jump is used to exit from the body of a loop, the target label is the first statement after the loop. The jump can be changed into a *break* statement.



(Original) (Transformed)

Figure 5. Break_goto pattern

while_goto : in Figure 6 a sequence of instructions `<stmt_seq_0>` is preceded by a conditional jump (`goto L1;`), if the condition holds the sequence is skipped. The last instruction of the sequence is a jump (`goto L0;`) to the previous conditional jump. This way, two jumps are used to simulate the behavior of a *while* statement. The code pattern can be changed accordingly.



(Original) (Transformed)

Figure 6. While_goto pattern

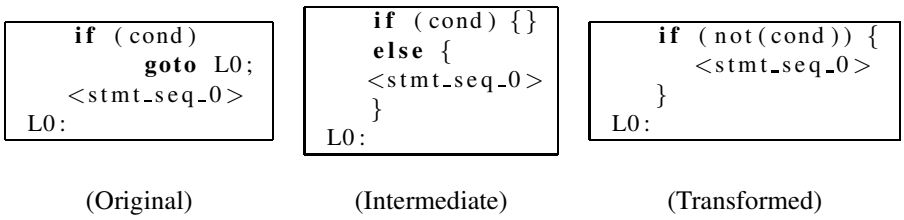


Figure 7. If_goto pattern

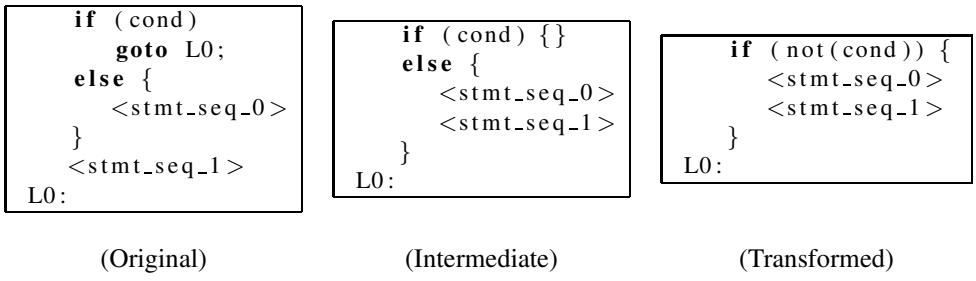


Figure 9. Ifelse_goto pattern

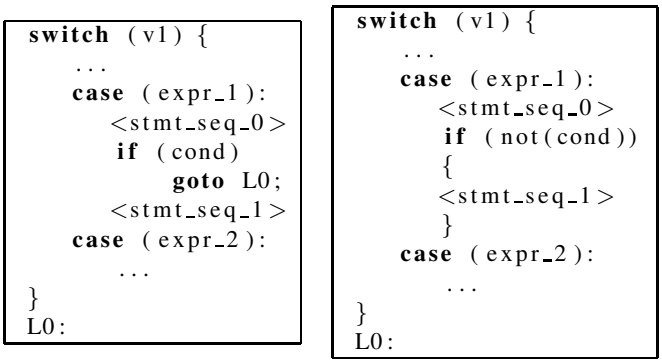


Figure 8. Goto_within_switch pattern

do-while_goto : a variant of the previous pattern, the only difference being that the involved statement sequence is executed at least once, even when the exit condition (`cond`) is true. This pattern can be transformed into a *do-while* cycle. This pattern is described in detail above and is shown in Figure 1.

2.2 Bohm-Jacopini top level

One of the first approaches to goto elimination was proposed by Bohm and Jacopini [3]. Their contribution is quite theoretical and consists of a set of transformation rules that should be applied to normalize the program flow graph. From these rules a transformation algorithm can be derived as shown in Figure 10. The transformation aims at making the program counter explicit in a variable. Jumps are turned into assignments to the program counter variable. What code to execute depends on a *switch-case* statement on the value of the program counter.

The steps required to achieve this transformation are the following:

- Some new variables are introduced:
  - the program counter is made explicit in the local variable `pc`;
  - the original label *a* is changed into the local constant `a`;
  - a constant `done` is introduced to handle the exit condition.
- The code is broken into segments according to labels. In the example, two segments can be identified: before the label *a* and the rest of the code from the label to the end.
- A while loop is added. The loop is executed until the value of `pc` becomes the exit value (i.e., `done`).
- The body of the loop is a *switch* on the value of the program counter `pc`, a *case* is added for each value from 0 to `done`-1. Each *case* body is concluded by two common instructions, the increment of `pc` value and a *continue* to the next *while* loop iteration.

- Eventually, *goto* statements are replaced by two instructions: an assignment to the `pc` and a *continue*.

```
    if (c)
        goto a;
    <stmt_seq_1>
a:  <stmt_seq_2>
    <stmt_seq_3>
```

(Original)

```
    int pc = 0;
    const a = 1;
    const done = a+1;
w:  while (pc != done) {
        switch (pc) {
        case 0:
            if (c) {
                pc = a;
                continue w;
            }
            <stmt_seq_1>
            pc++;
            continue w;
        case 1:
            <stmt_seq_2>
            <stmt_seq_3>
            pc++;
            continue w;
        }
    }
```

(Transformed)

Figure 10. Bohm-Jacopini strategy

This transformation algorithm is able to eliminate all occurrences of unstructured jumps. However, in the presence of jumps to labels on a different level of nesting, the result is expected to be quite tangled and difficult to understand, because the nesting structure is completely destroyed by the algorithm in such cases. However, when jumps refer to top-level labels (as in Figure 10), the quality of the resulting code is acceptable. Programmers judged it sufficiently recognizable as a transformation of their original code. So, we applied this transformation only when the *goto* statement points to a label that has nesting level equal to zero (top level).

2.3 Erosa

The goto-elimination strategy proposed by Erosa [8, 7] allows eliminating all gotos in a program, including irreducible gotos. Their approach has been developed to implement the preliminary step of an optimizing/parallelizing compiler that requires a structured control flow in order to perform particular optimizations.

They propose two key transformations, called *goto-elimination* transformations, devoted to remove a *goto* statement that appears at the same level of nesting as the target label. Other transformations, called *goto-movement* transformations, must be applied to prepare the code. A goto is moved either inward or outward, according to its nesting level, until it appears at the same level of the corresponding label.

```
    <stmt_seq_1>
    if (cond)
        goto L0;
    <stmt_seq_2>
L0: <stmt_seq_3>
```

(Original)

```
    <stmt_seq_1>
    if (!cond)
        <stmt_seq_2>
L0: <stmt_seq_3>
```

(Transformed)

Figure 11. Erosa goto-elimination with conditional

```
    <stmt_seq_1>
L0: <stmt_seq_2>
    <stmt_seq_3>
    if (cond)
        goto L0;
```

(Original)

```
    <stmt_seq_1>
    do {
L0:     <stmt_seq_2>
        <stmt_seq_3>
        }
    while (cond);
```

(Transformed)

Figure 12. Erosa goto-elimination within loop

In order to apply this technique, any jump instruction must be conditional to an *if* statement. All *goto*s can be easily reduced to this case, by adding a fictitious *if* statement with a condition that always holds.

Goto-elimination transformations Which *goto-elimination* transformation to apply depends on the position of the *goto* statement with respect to the corresponding label. If the *goto* precedes the label the case in Figure 11 applies. In this case, if the condition holds, execution jumps ahead, skipping some instructions (i.e., `stmt_seq_2`). This *gotos* can be easily removed by moving the skipped statements in the true-branch of the *if* statement, after negating its condition.

In case the *goto* follows the corresponding label (see Figure 12), a jump back happens when the condition `cond` holds. The same semantics is guaranteed by a *do-while* loop that contains all the statements originally placed between the label and the *goto* (i.e., `stmt_seq_2` and `stmt_seq_3`). The loop condition of the cycle corresponds to the original *if* condition.

Goto-movement transformations The *goto-movement* transformations apply when the jump instruction and the corresponding label are not at the same nesting level. Sev-

eral transformations are defined to move the *goto* statements across nesting levels, depending both on the movement direction (inward and outward) and on the statements to traverse (*switch-case*, *if-then-else*, *while*, *for*).

The case of an outward movement outside of a *switch-case* statement is shown in Figure 13.

- A brand new `goto_L0` variable is introduced and assigned the boolean value of the condition that controls the *goto* to remove.
- The goto is replaced by a *break* statement.
- The goto is moved just after the end of the *switch-case* body, but it is controlled by the value of `goto_L0`.
- Before executing the instruction at the given label, the value of `goto_L0` is reset to false.

The addition of a new variable avoids the evaluation of the same condition twice, keeping the behavior of the original code. Such an evaluation, in fact, could have an undesired side effect in case of duplicated evaluation.

The movement and the elimination of a *goto* statement could affect the position of the other gotos. For this reason, the algorithm by Erosa proceeds up to the elimination of the target goto, before considering the next one. Moreover, Erosa [7] claims that different final code could be obtained, depending on the order in which *goto*s are eliminated. In our migration project, we considered two different orders, based on how *goto* statements appear in the parse tree:

- backward: bottom-up; and
- forward: top-down.

2.4 JGoto

The last strategy is not actually a goto-elimination strategy. It relies on a post-compilation Java byte-code transformation, in order to support goto statements in Java. The strategy is changing the target language instead of changing the program to be migrated. This is a backup strategy that makes sense whenever each alternative goto-elimination strategy fails to deliver maintainable code.

When the legacy code is transformed to Java, *goto* statements and labels are replaced by placeholder invocations of stub methods: respectively, `jgoto("label")` and `jlabel("label")`.

After compilation, the resulting Java byte-code is subjected to transformation. Every invocation to `jlabel` becomes an automatically generated Java bytecode label, associated with the string appearing as the actual parameter. Every invocation to the placeholder method `jgoto` is replaced by an actual *goto* (supported in the Java byte-code) to the label associated with its string parameter.

2.5 Tool support

The first three strategies have been implemented using the TXL [5] language. TXL supports the definition of grammar–based rules to perform code transformation. Rules are divided into two parts, the pattern to be matched and the replacement. Transformations can be naturally coded in TXL staring from the code samples shown in this section.

The JGoto transformation tool was implemented by means of ASM,[1] a byte-code manipulation library. An example of the bytecode transformation operated by ASM to support gotos in Java is provided in Figure 14.

3 Results

In this section we present some empirical data collected during the execution of the goto-elimination task in our migration project. We interpret such data, trying to derive reusable lessons from them.

3.1 Metrics

All the strategies have been applied to the legacy system under migration and the resulting code has been analyzed in order to determine the most appropriate solution for the goto problem.

The first metric we used is the percentage of *goto* statements that each strategy is able to remove.

The migrated code should be easy to understand and it should be as similar as possible to the original system. An undesired side effect of some of the goto elimination algorithms considered is the increase of the nesting depth of the code. The presence of deeply nested control statements might represent an obstacle to program understanding which, in the extreme case of tens or hundreds of nesting levels, makes the code completely unmaintainable. On the other hand, to restructure complicated and tangled jump instances, a deeply nested code structure might be the only possible solution. We measured the quality of the resulting code as the maximum level of nesting of control statements that occur in a file (in the following, *max-depth* metric).

After *max-depth* has been calculated for each source file, a derived metric can be obtained, the *mean-max-depth*, i.e., the average of the max-depth values over the files in the system under migration.

The last metric we used to evaluate the outcome of goto elimination is the *time* taken by our implementation of each strategy to carry out the elimination. We measured how many hours the transformation of the whole system (8 MLOC) takes on a high-performance computer composed

[1] http://asm.objectweb.org/

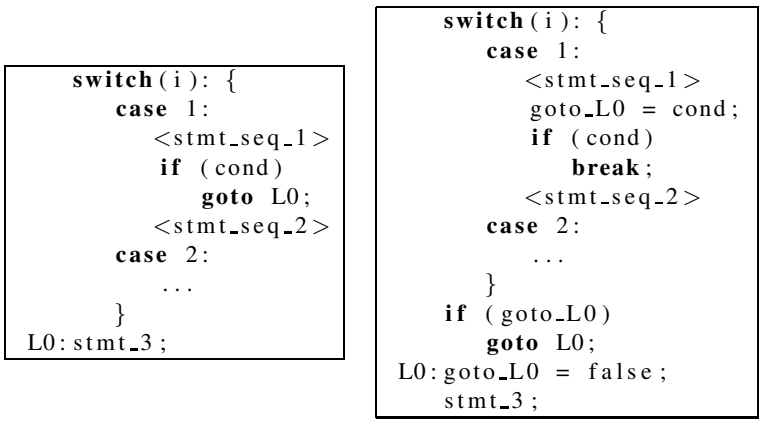


Figure 13. Erosa goto-movement transformation: moving a goto out of a switch-case

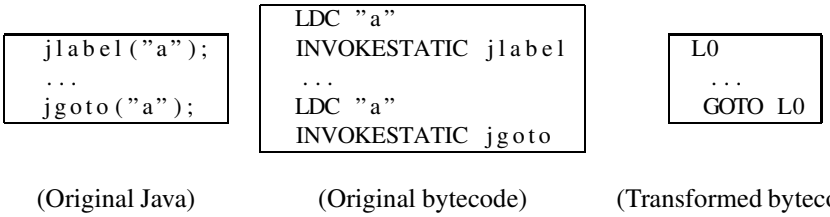


Figure 14. Bytecode transformation to support gotos in Java

of four nodes running Linux. Each node contains two dual-core processors (Intel Xeon 3.00GHz) and 4Gb of ram.

3.2 Empirical data

After macro expansion, the legacy system consists of 10,988,103 lines of code distributed over 13,195 source files. This code base contains 531,351 *goto* statements, or a mean density of one *goto* every 21 lines. The original code presents a maximum *max-depth* of 28 and a *mean-max-depth* of 3.07.

Table 1 reports detailed results of the pattern-based strategy. The second column reports how many times each pattern has been applied, the percentage is the ratio between the *goto*s eliminated with the given pattern and the total amount of *goto*s eliminated with the pattern-based strategy.

The most adopted pattern (34.3%) is associated with the *continue* statement, the most recently introduced new keyword in the original programming language. This could mean that a significant amount of the current code was developed when such a construct was not yet available, or that the new construct is not yet widely used.

The high number of *useless_goto* (13.6%) does not mean that in the original code there were many trivial *goto*s, pointing to the next line. It means, instead, that this pattern appeared in the code as a side effect of the other transformations.

Pattern	Occurrences	
useless_goto	15,397	13.6%
if_goto	14,145	12.5%
if_goto_then	6,028	5.3%
goto_within_switch	486	0.4%
ifelse_goto	175	0.2%
continue_goto	38,888	34.3%
break_goto	14,378	12.7%
while_goto	20,562	18.2%
dowhile_goto	3,236	2.9%
TOTAL	**113,295**	

Table 1. Results of pattern based strategy

It should be noted that the simplest patterns (i.e., *useless_goto*, *if_goto*, *continue_goto*, *break_goto* and *while_goto*) were applied more often that the more complicated ones (i.e., *if_goto_then*, *goto_within_switch*, *ifelse_goto* and *dowhile_goto*).

Table 2 compares the results of the adoption of all the

Strategy	Goto removed	Max-depth	Mean-max-depth	Time
Pattern-based	21%	29 (+1)	3.48 (+13%)	1h 2”
Bohm-Jacopini TL	79%	30 (+2)	4.48 (+46%)	22”
Erosa (backward)	100%	162 (+134)	10.18 (+232%)	117h 38”
Erosa (forward)	100%	207 (+179)	14.78 (+381%)	144h 22”
JGoto	0%	28 (+0)	3.07 (+0%)	–

Table 2. Results in the adoption of the considered goto-elimination strategies

considered strategies. The pattern-based strategy reached a relatively low rate of elimination (21%), but produced high quality code, with nesting levels comparable to the original ones. Bohm-Jacopini top-level (TL) performed better (79%) than patterns, in terms of number of gotos eliminated, still with a good quality of the resulting code.

As expected none of these two strategies was able to achieve complete elimination. However, the resulting code quality suffered just a minor impact, which justifies their usage in practice. The maximum level of nesting remains more or less the same (+1 and +2 respectively). The *mean-max-depth* increases in a slightly more sensible way (+13% and +46%)

Erosa is known to be able to eliminate all gotos in the code, including irreducible jumps. We applied Erosa both in backward and forward mode. In both cases, it caused an unacceptable increase in the maximum (+134 or +179) and mean (+232% or +381%) level of nesting. Moreover, the amount of time needed to apply this strategy is considerably higher than the pattern-based and Bohm-Jacopini TL strategies.

The strategy based on JGoto allows full preservation of the code quality, but it does not remove any *goto*. As a backup strategy, it is supposed to be used only if the selected strategies do not achieve complete goto removal and if the remaining gotos represent a substantial amount that cannot be handled manually. The costs of this operation are not in code transformation within the original programming language. In fact, this strategy can be applied only during translation to Java, hence its cost is partially embedded into the translation costs. Some extra cost is paid whenever the migrated system is compiled, since the generated bytecode has to be processed by our ASM bytecode transformer which replaces static invocations to `jlabel` and `jgoto` with the corresponding bytecode constructs. Our preliminary estimates indicate a negligible compilation overhead due to the bytecode transformation made by ASM. Another penalty is paid at runtime, because compiler optimizations must be disabled when producing the bytecode, so that the pattern manipulated by the ASM tool remains recognizable and uncorrupted. However, just in time optimizations that are carried out dynamically remain possible. In Table 2, no code transformation time is reported, since the original code is left untouched and the strategy comes into play only when migrating the original code to Java.

4 Discussion

Considering the data reported in the previous section, adoption of the Erosa strategy is not feasible. In fact, it is not possible for the developers to maintain a code base with nesting level greater than one hundred. Such a nesting level makes any code change impossible or highly likely to introduce bugs. Comprehension of such code was judged as a “mission impossible” task by programmers. Familiarity is completely lost, as well as resemblance of the transformed code with the original one. Although we do not have data to claim it, we expect that any general purpose goto elimination strategy that is capable of removing all gotos in the code produces unmaintainable and unfamiliar code, once applied to the legacy system being migrated in our project, because of the tangled control flow structure implemented in the code.

At the other extreme, leaving all *goto*s in the code and using the JGoto strategy to render them in Java preserves nesting level and familiarity, but it misses the opportunity of improving the code structure in the ongoing migration process. The resulting code would be perfectly readable and maintainable for the current programmers of this system. It would be regarded as ugly Java code with uncommon and unfamiliar structure by any new Java programmers hired by the software company. No Java programmer would probably ever organize the code in that way. Moreover, the resulting code after migration would not be “true” Java code, since it contains placeholder method invocations (`jlabel(), jgoto()`), which alter the intraprocedural control flow. The result would be Java code with gotos.

The pattern-based strategy and the Bohm-Jacopini TL strategy are partial, in that they leave some goto statements in the code, but they both produce quite good code. The main strength of the pattern-based strategy is its small impact on the code. The result was regarded as an improved version of the original code by the programmers. The Bohm-Jacopini TL strategy offers wider coverage (it eliminates more gotos), but still with an acceptable impact on the code quality.

Based upon the considerations made above, we discarded Erosa and decided to apply the pattern-based strategy and the Bohm-Jacopini TL strategy. The *goto*s that still remain in the code will be kept when migrating to Java, by adopting the JGoto strategy. However, JGoto represents a temporary solution, because the elimination of the migrated *goto*s remains an important long term goal. They will be manually removed, as soon as the various modules would undergo ordinary evolution. More generally, we expect the migrated code to keep several links with the original programming language and execution environment. Another example is the data model. The Java classes obtained from the migration of the existing data structures often deviate from the principles of "good" object-oriented design. For instance, to support multiple views on the same data, the functional equivalent of a *union* is produced by the translator. However, a long term goal is removing such links with the past and evolving the migrated Java code so as to become truly object oriented and Java style.

The goto-elimination task that we executed on the legacy system under migration gave us the opportunity to consider some relevant aspects of reengineering that arise when dealing with large, complex legacy systems. The lessons that we learned can be generalized as follows:

1. Assessment of the output of migration involves multiple dimensions that often address conflicting needs. One instance of such a dichotomy was removing all gotos vs. producing maintainable code.

2. Given a complex system, no single solution is expected to solve *all* instances of the problem to be addressed. The variety of cases that can be present in the system imposes the evaluation and combination of multiple alternatives.

3. Solutions that are appealing for their theoretical properties (e.g., elimination of all gotos with a number of extra variables linear with the number of labels and no code duplication, as offered by Erosa) might reveal inappropriate in practice, because of the specific features of the system under migration.

4. Backup solutions (such as JGoto) are important to let the project proceed even when major obstacles are encountered. Although they may introduce some "code smells", they are a milestone toward the "ideal" solution.

5. Migration projects should have short term and long term goals. Not everything can be achieved in the short term and often compromises are necessary to be able to deliver the expected results. However, a long term strategy should also be defined, so that improvement continues even when migration is over.

5 Related works

Criticisms on the usage of unstructured constructs have been raised for the first time forty years ago by Dijkstra [6]. This had a major impact on the scientific community and, immediately, Dijkstra's view about *goto* statements became widely accepted. However, later, a debate has been started by opponents [13] and supporters [9] of Dijkstra's position.

Since the *goto* problem has been identified, several techniques have been proposed to remove them and obtain structured code. The most relevant for this paper are the work presented by Bohm and Jacopini [3] and by Erosa [8, 7]. Bohm and Jacopini [3] formally showed that any algorithm that can be implemented using *goto*s can be also implemented by using structured programming. Instead of providing a transformation algorithm, they proposed a set of patterns and transformation rules to normalize a flowgraph. A practical implementation has been provided later by Peterson [11] based on node splitting to transform any flowchart into a well-formed flowchart (corresponding to a program with only *if*s and *loop*s properly nested). A similar approach has been used by Williams [16, 17] based on the identification of some known unstructured sub-graphs. Unstructured sub-graphs are then changed into their equivalent structured graphs, i.e. graphs that can be decomposed into sequences, selections and loops. Finally, an alternative approach to node splitting has been proposed by Ashcroft [1]. It relies on introducing fresh program variables and on the usage *while* loops.

The method proposed by Erosa [8, 7] takes advantage of program transformations. Her work was originally devoted to implement an optimizing compiler, but it has more general applicability. The transformation is performed in two steps, a preliminary normalization and a successive elimination of *goto*s. In the normalization, the goto is moved through the code until it reaches the same level of nesting as its target label. After that, the *goto* is replaced by an equivalent structured statement. Erosa presented a detailed list of transformations for both phases, showing how the program semantics is maintained through the process.

A relevant approach has been also presented by Baker [2] to eliminate most of the unstructured statements in Fortran code. Her purpose was quite different from ours. She was interested mostly in improving the readability of the code, even when this entails major code changes (restructuring), which are allowed in her context. We are, instead, interested in a final code that is as similar as possible to the original one (familiarity preserved). An approach that implies major code changes has also been proposed by Cifluentes [4], to recover understandable source code starting from binaries.

Two large industrial legacy systems (5.2 Mloc in total) have been subjected to goto-elimination by Veerman [15]. Similarly to us, his approach was based on patterns, in par-

ticular, the ones proposed by Sellink [14] for COBOL/-CICS. However, while we require high similarity with the original code, Veerman's approach leads to major changes, that improve maintainability. In contrast to us, he does not allow the introduction of novel variables, but he allows code duplication (to a limited extent).

Mueller [10] allows code duplication to remove unstructured statements. This solution is not viable for us because it involves major code modifications, with a potential explosion of the already large size, and because the understandability of the final result would probably be not acceptable. Ramshaw [12], instead, is willing to eliminate *goto*s while preserving the program structure. He showed that this task can be achieved without code duplication only in case the program enjoys the reducibility property (no irreducible jump present).

A trade off between code replication and conditional expression complexity has been presented by Zhang and D'Hollander [18]. Their approach relies on the concept of hammock graph, a subgraph with a single entry point and a single exit point.

To our knowledge, no prior work presented a real assessment of alternative goto elimination strategies on a large industrial case study, where different goto-elimination approaches are compared in terms of maintainability of the resulting code.

6 Conclusion and future work

The problem of removing unstructured programming constructs, no longer supported by most modern programming languages, has been broadly addressed in the past. In this paper we presented our experience in eliminating *goto* statements from a real, large legacy system, in the context of its migration to Java.

We compared the results of different strategies in terms of completeness and resulting code quality. Complete solutions turned out to be not feasible because of the poor quality of the final code. In practice, the most appropriate solution was to trade completeness for higher code quality. A general lesson that we learned in the execution of the goto elimination task is that compromise, partial solutions are often necessary to meet the conflicting goals of a migration project, and long term objectives should be set to address the problems for which only backup solutions are viable in the short term.

In the future, we will face the next steps of the migration project. Several challenging problems will be addressed, such as the transformation of the persistent data model from indexed ISAM files to relational database, translation of existing data structures to Java classes, and migration of the user interface from character oriented to GUI.

References

[1] E. Ashcroft and Z. Manna. Translating Program Schemas to While-Schemas. *SIAM Journal on Computing*, 4:125–146, 1975.

[2] B. Baker. An Algorithm for Structuring Flowgraphs. *Journal of the ACM (JACM)*, 24(1):98–120, 1977.

[3] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM*, 9(5):366–371, 1966.

[4] C. Cifuentes. A structuring algorithm for decompilation. *Proceedings of the XIX Conferencia Latinoamericana de Informatica*, pages 267–276, August 1993.

[5] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, 2002.

[6] E. W. Dijkstra and D. Questions. Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[7] A. M. Erosa. *A Goto-Elimination Method And Its Implementation For The McCat C Compiler*. PhD thesis, McGill University, Montreal, School of Computer Science, May 1995.

[8] A. M. Erosa and L. J. Hendren. Taming control flow: a structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229 – 240, May 1994.

[9] D. Moore, C. Musciano, M. J. Liebhaber, S. F. Lott, and L. Starr. "goto considered harmful' considered harmful' considered harmful? *Communications of the ACM*, 30(5):351–355, 1987.

[10] F. Mueller and D. Whalley. Avoiding unconditional jumps by code replication. *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 322–330, 1992.

[11] W. Peterson, T. Kasami, and N. Tokura. On the capabilities of while, repeat, and exit statements. *Communications of the ACM*, 16(8):503–512, 1973.

[12] L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM (JACM)*, 35(4):893–920, 1988.

[13] F. Rubin. 'goto considered harmful' considered harmful. *Communications of the ACM*, 30(3):195–196, 1987.

[14] A. Sellink, H. M. Sneed, and C. Verhoef. Restructuring of cobol/cics legacy systems. *Science of Computer Programming*, 45(2-3):193–243, 2002.

[15] N. Veerman. Revitalizing modifiability of legacy assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003*, 16(4–5):219–254, 2004.

[16] M. Williams. Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal*, 20(1):45–50, 1977.

[17] M. Williams and H. Ossher. Conversion of Unstructured Flow Diagrams to Structured Form. *The Computer Journal*, 21(2):161–167, 1978.

[18] F. Zhang and E. H. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, 30(4):231–245, 2004.