Deciding between if and switch when writing code (part 1 of 2)

Experiment performed at the 2009 ACCU Conference
First published in CVu vol. 21 no. ?

Derek M. Jones Knowledge Software Ltd derek@knosof.co.uk

1 Introduction

When writing software a common requirement is for the execution of some sequence of statements to depend on a variable having a particular value. Programming languages provide various constructs to support this requirement, e.g., the if-statement (which often supports checking against a single value) and the switch-statement (which supports the checking against a set of values). Measurements show that approximately every fifth statement is a selection statement.

This article investigates the possible factors that influence developers in their choice of selection statement, i.e., when deciding whether to use an if-statement or a switch-statement to implement some desired functionality. Two sources of data are analysed: measurements of existing code and the results of an experiment carried out at the 2009 ACCU conferences.

This article has two parts, the first (this one) discusses measurements of **`if`** and **`switch`** statement usage in C source code, looking for differences in usage patterns; while a second one analyses the results of an experiment that asked subjects to write code whose behavior depended on a variable that could take multiple values.

Language support for a switch-statement is unnecessary in the sense that it is always possible to write a sequence of if-statements that achieves the same effect. Reasons for supporting a switch-statement include ease of compiler optimizations[5] (i.e., for two equivalent sets of selection statements significantly less compiler implementation effort is needed to generate good quality code) and the belief that use of the **`switch`** form requires less developer effort and in some circumstances is less error prone.

As an example the following code (referred to as an *if-else-if* sequence, the expression appearing between parenthesis is the *controlling expression* and `var` is the *tested-expression*) where `C_1` and `C_2` are compile time constants:

```
1  if (var == C_1)
2     stmt_seq_1;
3  else if (var == C_2)
4     stmt_seq_2;
```

could be written as (provided none of the blocks in the above sequence contained a **`break`** statement; an **`else`** appearing in the second if-statement would be mapped to a **`default`**):

```
1  switch (var)
2     {
3     case C_1: stmt_seq_1;
4               break;
5
6     case C_2: stmt_seq_2;
7               break;
8     }
```

Common constraints on the use of the switch-statements include: the value must be known at translation time and that the value be representable as an integer type. Languages that do not have one or more of these constraints include PERL (using the `Switch` module) where the value need not be constant and can have any type for which equality is defined, and C# supports the use of string literals in case-labels.

Here we will limit ourselves to the situation where case-label values must be constant and representable in an integer type. Languages that have these constraints include Java, C and C++.

2 Application, algorithmic and evolutionary factors

The runtime execution decision represented by an **`if`** and **`switch`** statement is a consequence of an application or algorithmic requirement. An example of an application requirement is the handling of user input generated by the selection of an item from a list of options displayed by a program. An example of an algorithmic requirement is checking whether a value is within the bounds supported by the algorithm.

Almost no information is available on the break down between application and algorithmic requirements. Work on the Model C Implementation provides one data point for the break-down of applications vs. algorithmic if-statement usage. Every if-statement was tagged as being either as an application requirement (i.e., a requirement specified in the C Standard) or an algorithmic requirement.[2] Of the 4,329 if-statements (excluding the contents of the support library directories) 54.3% were tagged as applications requirements. This implementation differed from most compilers in that it performed no optimizations and so is likely to underestimate the percentage of if-statements attributable to algorithmic requirements. No break down by controlling expressions involving equality tests against constants was reported.

At a particular point in the code factors such as the number of conditionally executed statement sequences, the number and kind of values tested and the amount of code that depends on the decision made are likely to be outside the immediate control of the developer writing the code. The developer simply gets to decide how to write the code.

As it is being written code evolves and code that is part of an application that is used often evolves over a longer period than it took to originally write. The factors driving a developer's decision making process may be different between initial development and maintenance. During initially development a switch-statement might be chosen if the author anticipates that the code will soon be updated to include additional sequences of conditionally executed statements. During maintenance statements will be added and removed and the remaining code may be left untouched or effect the kind of code that is added. For instance, when extra if-statements are added to an existing if-else-if sequence does is the combined code refactored as a switch-statement, or when case-labeled statements are removed from a switch-statement is the code rewritten as an if-else-if sequence?

Some coding guidelines recommend that **`default`** and **`else`** always be used, the intent being to catch unanticipated out-of-bounds conditions.

2.1 Factors influencing developer choice

The following list are some of the factors that might influence a developer's decision on whether to use an **`if`** or **`switch`** statement:

- Cognitive biases. Does a developer carry out enough analysis before writing code to have a reasonable idea of what is involved, then using this information to decide which selection statement is most appropriate, or does a developer use fast and frugal heuristics[1] or is the choice more stream of consciousness driven? Specific possibilities include:
  - laziness, such as not investing the effort needed to accurately estimate the likely structure of the code being written or the perceived physical effort needed to write the code, e.g., amount of typing.
  - unwillingness to change a course of action that has been embarked upon (i.e., a developer starts using an if-statement, because that is the common case, and continues to use it after discovering that multiple tests are involved).
  - the current frame of mind. This might result in one kind of statement being used because it is written immediately after having written the same kind of statement, i.e., an if-statement will be preferred after another if-statement and a switch-statement preferred after or within a switch-statement.
  - existing practices on the use of selection statements, i.e., what is commonly seen in source. For instance, while it is not incorrect to use a switch-statement where a single if-statement would suffice, based on existing practice such usage might be considered *unnatural* by developers. The opposite situation where a series of if-else-if-statements are used where a switch-statement could have been used is perhaps not viewed with the same degree of surprise.

- Developer expectations on the number of conditional arms expected to occur in the final code. Perhaps the probability of switch-statement being preferred increases as the number of arms expected increases.

- Developer are often driven by a desire to write efficient code and they have beliefs about whether a compiler is likely to generate higher quality code for one construct compared to another. The perceived runtime overhead of the expression being compared against may cause an increase in the probability of a switch-statement being preferred as this perceived overhead increases.
- Developer expectations on the number of different values that are likely to be compared against within the controlling expression of each conditional arm (e.g., three comparisons are needed for an arm that is executed if its tested-expression equals 4, 5 or 6). Perhaps the probability of switch-statement being preferred increases as the number of comparisons that are expected to be made increases.
- Developer expectations on the number of statements likely to be present in the conditionally executed arms.

3 Analysis of existing source

Measurements of **`if`** and **`switch`** statement usage in existing source can provide evidence about whether certain factors are likely to influence developer choice. This subsection discusses measurements of various kinds of **`if`** and **`switch`** statement usage in the visible form of a number of large C programs (e.g., gcc, idsoftware, linux, netscape, openafs, openMotif and postgresql).

The primary tool used to make these measurements was Coccinelle.[4] This tool converts the visible form of C source to an abstract syntax tree and provides a mechanism to specify patterns that match against this representation. The following is an extract from a pattern that matches a sequence of if-else-if statements, storing information on the position (line and column) of various constructs; other parts of this pattern write out the matched expressions `E_1` and `E_2` and their locations.

```
1  expression E_1, E_2;
2  statement S_1, S_2, S_3;
3  position p_1, p_2, p_3, p_4;
4  @@
5  if@p_1 (E_1)
6     S_1
7  else@p_3 if@p_2 (E_2)
8     S_2@p_4
9  else
10    S_3
```

Conditional preprocessing directives, e.g., **`#if`**/**`#endif`**, are a significant source of problems for tools attempting to parse the visible source. Coccinelle is able to parse source containing these directives provided the conditional arms contain complete statements, declarations or expressions (measurements have shown this to be the majority of instances[3]). The version of Coccinelle used (0.1.10) internally handles conditional compilation directives in a way that causes some patterns to ignore selection statements containing such directives. The patterns used to measure statement counts (see Figure .4) are the only ones known to be affected by this behavior.

3.1 if-statement characteristics

This subsection describes the process used to extract information about if-statement sequences that could be mapped to an equivalent switch-statement.

3.1.1 Constants

Most of the usage patterns being searched for require the controlling expression to contain one or more equality tests against a constant value. Symbolic names are often used to denote constant values in the visible source, these symbolic names might be defined as macros or enumeration constants. Macro names do not usually contain lower-case letters[3] and Coccinelle has the ability to treat identifiers that don't contain any lower-case letters as having a constant value.

To validate the accuracy of constant detection two sets of Coccinelle patterns were written, one requiring that at least one operand be a constant and the other having no such requirement. Comparing the output of the constant and non-constant pattern (7,727 non-constant occurrences for if-else-if and 2,742 occurrences for if-if), the constant pattern matched 77% (if-else-if) and 82% (if-if) of all possible matches. A manual check of the non-constant cases showed that while some were constant, but not treated as such because their name included lower-case letters, the number was sufficiently small that they could be ignored.

`NULL` usually denotes the null pointer constant, which is not a valid value in a case-label. Any if-statement sequence whose tested-expression was compared against this symbolic value was excluded from these measurements. In practice if-statement sequences containing this constant generally only occurred for sequences that involved a single tested-expression, see Figure .2.

3.1.2 Partial sequences

An if-else-if sequence may contain within it a subsequence that has the characteristics required for it to be mappable to a switch-statement. For instance in the following:

```
1  if (var == C_1)
2     stmt_seq_1;
3  else if (var == C_2)
4     stmt_seq_2;
5  else if (expr_1 != non_constant)
6     stmt_seq_3;
```

the first two controlling expressions have the desired characteristics, but the third does not. Could this sequence be reordered so that the two expressions with the desired characteristics appeared last and so could be mapped to a switch-statement? Answering this question for most if-else-if sequences requires resources not available on this project and such subsequences were not included in the measurements reported here.

Only those if-else-if sequences whose controlling expressions all have the desired characteristics or those where the appropriate controlling expressions occurred last were included in the measurements reported here.

An if-if sequence differs from an if-else-if sequence in that replacing one of its subsequences by a switch-statement will not effect the control flow of any if-statements that appear immediately before or after it. The right plot of Figure .1 includes any mappable if-if subsequence, while the left plot does not count an if-if sequence if it is immediately preceded or followed by a non-mappable if-statement.

3.1.3 if-else-if sequences

The process used to extract if-else-if sequences that are mappable to a switch-statement was as follows:

1. The controlling expressions in all if-else-if sequences were extracted. For instance, the three expressions `expr_1`, `expr_2` and `expr_3` would be extracted from the following code:

   ```
   1  if (expr_1)
   2     stmt_seq_1;
   3  else
   4     if (expr_2)
   5        stmt_seq_2;
   6     else
   7        if (expr_3)
   8           stmt_seq_3;
   ```

2. Those controlling expressions in each sequence that all had one of the forms: equality test against a constant, a series of such equality tests combined using the logical-OR operator or a *between* operation implemented using relational operators and the logical-AND operator (combinations such as the constant appearing on the left-hand side and other ways of expressing *between* were checked for) were extracted. These expressions map to case-labels as follows:

```
expr == constant                          ⇒ case constant:

expr == constant_1 || expr == constant_2 ⇒ case constant_1:
                                             case_constant_2:

expr >= min_const && expr <= max_const   ⇒ case min_const:
                                             case ...
                                             case max_const:
```

where `expr`'s token sequence is identical in every expression within the sequence (e.g., the expressions `x+y` and `y+x` were considered to be different).???

Testing whether an expression having an unsigned type is less than some small constant is effectively a between operation. There was not sufficient time to measure this usage.

3.1.4 if-if sequences

A sequence of if-statements of the form (referred to as an *if-if* sequence here):

```
1  if (x == 1)
2     stmt_1;
3  if (x == 2)
4     stmt_2;
```

is equivalent to:

```
1  if (x == 1)
2     stmt_1;
3  else if (x == 2)
4     stmt_2;
```

if the value of `x` is not changed by the execution of `stmt_1`, or the last statement of `stmt_1` is a **`return`** statement.
This equivalence also holds when any statements appearing between the two if-statements can be moved to before or after the sequence without changing the external behavior of the program. An optimistic search (the simplifying assumptions made are likely to overestimate the number of occurrences) for if-statements separated by short sequences of unrelated statements found a small number of possible instances. The number found is sufficiently small that it can be ignored without significantly effecting the results.
It was not practical to fully analyse the consequences of executing `stmt_1` to deduce whether it resulted in the value of `x` being modified. The number of occurrences of if-if sequences based on the following three levels of analysis measured, see Figure .1:

1. assuming that `stmt_1` does not modify the value of `x`.
2. detecting some of those operations that could result in the value of `x` being modified (e.g., assignment, having its address taken, pre/post increment); any called functions were not analysed to deduce their effect on `x`.
3. treating any occurrence of `x` in `stmt_1` as a modification of its value.

The compound statement associated with the first if-statement of an otherwise mappable sequence ended with a **`break`** statement in 1% of occurrences. The percentage of **`return`** statements appearing in this context was 5% for if-else-if sequences and 36% for if-if.

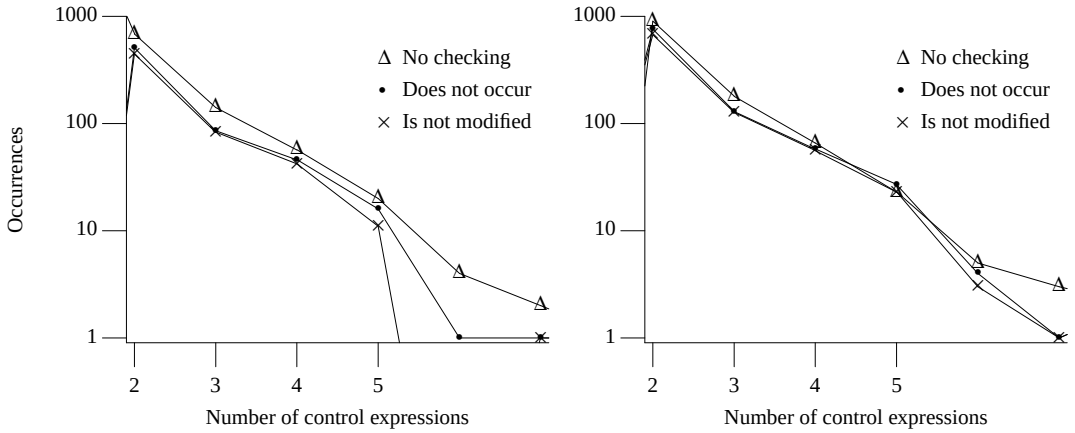

Figure .1: Number of if-if sequences of a given length at three levels of analysis. The left plot only includes those if-if sequences where all of the if-statements are mappable to a switch-statement. The right plot includes any if-if sequences that is a subsequence of a longer, mappable, if-if sequence. In the case of a two if-statement if-if sequence, taking the *no checking* measurements as representing 100%, the percentage of uses where the block associated with the first if-statement does not contain any of the (checked) operations that modify the tested-expression is 75%/84% (left plot/right plot); the percentage of such sequences where the first block does not contain an instance of the tested-expression is 65%/75%.

3.2 switch-statement characteristics

The following is an example of the structure of Coccinelle patterns used to extract information on switch-statement usage (a separate pattern matched code where the last case-labels in a switch-statement were not followed in a jump-statement):

```
1  expression E_1, E_2;
2  position p_1, p_2, p_3;
3  @@
4  switch (E_1)@p_1
5     {
6     case ...:@p_2
7               ...
8  (
9               break;@p_3
10 |
11              continue;@p_3
12 |
13              return;@p_3
14 |
15              return E_2;@p_3
16 )
17    }
```

This pattern matches any sequence of code, within a switch-statement, that starts with a case-label and ends with a **`break`**, **`continue`** or **`return`** statement.

Any case-labeled statements that *fall through* to the following case-labeled statement are treated as-if they consisted of the statements associated with the fallen-into case-labels. Falling through to another sequence of statements is sufficiently rare that its effect on these measurements can be ignored.

3.3 default and final else

Use of **`default`** in a switch-statement is equivalent to a final **`else`** in an if-else-if sequence and the two constructs should be counted in the same way.

A default-label is sometimes prefixed to the same statement sequence as one or more case-labels; in the source benchmarks measured for this paper 5.7% of all **`default`**s were so prefixed.[3] This usage can occur

through code evolution or a desire to explicitly map every named requirement in a specification to source code (this is sometimes handled via a comment). An example of this usage is:

```
1  case MEM_FAIL:
2  case DISC_FAIL:
3  default:        return ERR_CODE;
```

The nearest if-else-if sequence equivalent to the above code would be for the final controlling expression to perform equality tests on the case-label values and for the statement sequence in both of the conditional arms it controls to be the same. There are 25 instances where both arms of the final if-statement in a if-else contain the same statement sequence; a small number that has no significant effect on the measurement results.

3.4 Threats to validity

It is inevitable that the way in which selection statements are used will vary and some method of calculating the likely variation is needed if measurements of different constructs are to be compared in a meaningful way. For some of the measurements it was not possible to derive any statistically model and so no statistically meaningful comparisons can be made about the results obtained for those constructs.

Developers may make different decisions when writing new code compared to when modifying existing code. The C source measured for this study has been actively worked on for many years and during this time its selection statements are likely to have evolved. It is not possible to separate out any differences in this decision process for the measurements reported here (the experiment described in part 2 asked developers to write new code).

The only source code measured was written in C. To what extent is it possible to claim that the findings apply to code written in other languages? While there are no obvious reasons why the usage patterns found in C should not also occur in other languages, there is no reason why they should. Measurements of source written in other languages would help put this issue to rest.

The following are characteristics of the tool used, Coccinelle, and the patterns used that might be affected by these characteristics, i.e., the measurements might not be as accurate as expected:

- the patterns used to measure selection statement arm length only counted statements, i.e., they did not include support for the presence of declarations, consequently any selection statement containing declarations in its conditional arm was not counted. It is known that approximately 10% of locally defined objects are defined in nested scopes,[3] but the distribution of these definitions (i.e., whether the probability of them occurring increases as the number of statements in a block increases) is not known.
- the release of Coccinelle used (0.1.10) did not always match constructs containing conditional preprocessor directives. This effects the measurement of the number of statements contained within selection statements, resulting in those containing such directives being ignored.

4 Results

The measured source contains 29 times as many if-statements as switch-statements. Of the 384,749 if-statements measured 9.6% are contained within an if-else-if sequence and 15.9% of these sequences are mappable to a switch-statement. The corresponding values for the if-if sequence are 29.4% and 2.5% respectively.

There were 13,152 switch-statements measured and these contained a total of 63,395 **`case`** labels. A **`default`** label appeared in 49% of switch-statements.

While the percentage of all if-statements that include an **`else`** arm is 22%, the percentage of if-else-if sequences having a final **`else`** arm averages out at around 50% and the final if-statement in an if-if sequence contains an **`else`** arm in around 20% of occurrences (this percentage rapidly drops as the sequence length increases).

4.1 Cognitive issues

4.1.1 Typing effort

What is the difference in the amount of typing that needs to be done to create equivalent **if** and **switch** statements? In the following example underscores are used to denote characters that would appear in both forms of selection statement.
Comparing:

```
1  if (____ == __)
2     _______
3  else if (expr == __)
4     _______
```

with the equivalent switch-statement:

```
1  switch (____)
2     {
3     case __: _______
4              break;
5
6     case __: _______
7              break;
8     }
```

approximately twice as many non-whitespace characters occur in the switch-statement and depending on layout conventions there are also likely to be more than twice as many whitespace characters. The non-whitespace character ratio only comes down on the side of the switch-statement when the a **break** is not needed, i.e., a **return** or **continue** statement terminates most of the labeled statement sequences.

4.1.2 Effects of local context

When writing code to what extent does the last choice of selection statement made by a developer effect the probability that the same choice will be made next time a selection statement is written?
There was insufficient time available to perform the local context source measurements that could help provide an answer to this question.

4.2 Number of conditional arms in construct

How does the number of conditional arms in an if-else-if or if-if sequence compare with the number of case-labeled statement sequences that can be jumped to in a switch-statement?
The differences between the left and right plots in Figure .2 are caused by contributions from **else** and **default**. In the case of an if-if sequence the **else** can only appear on the final if-statement and a default-label be mixed with case-labels on the same statement sequence.
In Figure .2 the solid lines are a least-squares fit of the data to an exponential function;[1] **switch** fitted over the range 3 to 11, if-else-if fitted over 1 to 7 and if-if fitted over 2 to 7 give the following respective equations (with x being the number of conditional arms; rounded to two decimal digits):

$$switch \propto e^{3.92-0.16x} \tag{.1}$$

$$if\ else\ if \propto e^{4.96-0.58x} \tag{.2}$$

$$if\ if \propto e^{3.97-0.52x} \tag{.3}$$

These measurements suggest that the use of if-statement sequences decreases at approximately a fixed rate as the length of the sequence increases, with switch-statements taking up most of the slack for sequences of three or more.

[1] This data could have been fitted just as well with a power-law. Without a model describing developer behavior it is not possible to distinguish a power-law from an exponential function and the latter was used to keep out the baggage that usually accompanies the former.

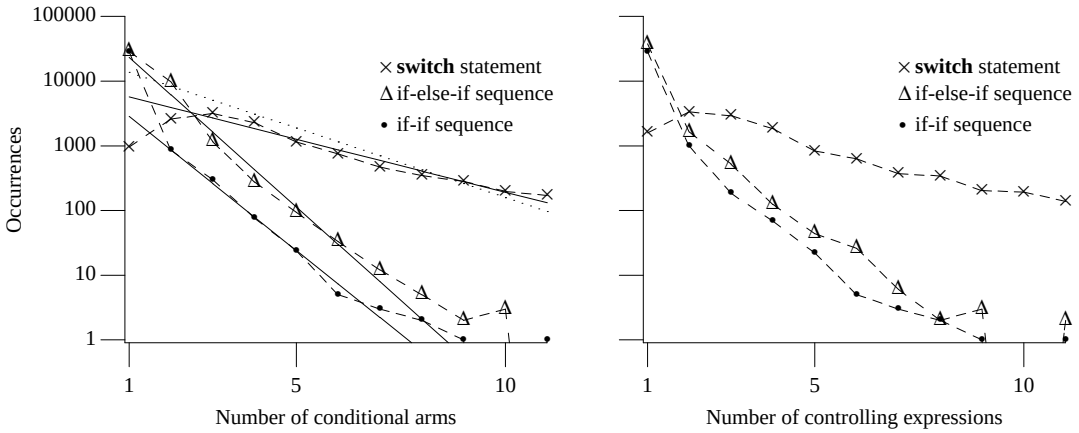

Figure .2: The left plot is of occurrences of if-else-if, if-if (uncertainty about which sequence a single if-statement, without an else-arm, belonged to was resolved by including it in both sequences) and switch-statements having the given number of conditional arms. The right plot is the number of controlling expressions in a if-else-if and if-if sequence and the number of case-labeled statement sequences (i.e., it excludes the effect of any **`default`**) in a switch-statement. The solid lines are a least-squares fit of the data to an exponential function. The dotted line is a fit to the sum of all sequences having a given number of arms.

The number of conditional arms that need to be written is likely to be controlled by factors that a developers' has no influence over. The dotted line in Figure .2 is a fit of the sum of all selection statements containing a given number of conditional arms, the equation is given by:

$$all \propto e^{4.35-0.22x} \tag{.4}$$

4.3 Expression runtime overhead

Function calls are generally perceived as having a high runtime overhead. Minimizing the number of function calls that need to be evaluated by the controlling expressions of a sequence of if-statements can be achieved either by assigning the result to a temporary variable that is then compared in each controlling expression or by using a switch-statement.

The percentage of all if-statement controlling expression containing a function call is 14%.[3] As Table .1 shows, the percentage of function calls in the selection statement sequences of interest in this paper is much lower. The percentage of function calls in switch-statement controlling expressions is higher than mappable if-else-if and if-if sequences. You author was not able to find a reasonable model that enabled the statistical significance of this difference to be estimated.

Table .1: Function calls appearing in the tested-expression of a controlling expression, as a percentage of the total number of occurrences of the associated selection statement.

Selection statement	Percentage containing function calls
switch	2.30%
if-else-if	0.47%
if-if	0.88%

For those expressions that do not include a function call the number of operators that need to be evaluated at runtime is one possible measure of perceived runtime overhead. Table .2 shows that approximately 90% of tested-expressions do not contain any unary or binary operators; this figure drops to 60-70% if the `->` operator is counted.

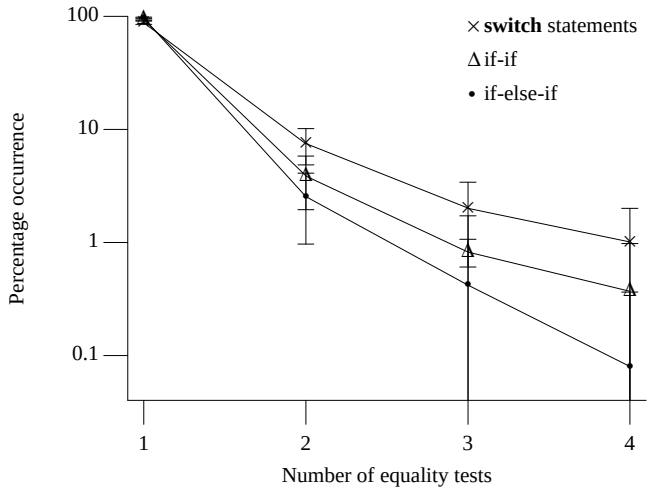

Figure .3: Percentage of equality tests performed in the control expressions of if-else-if (97% had one) and if-if sequences (95% had one), and the number of **`case`** labels appearing together on the same statement (88% had one; any **`default`** label was not counted). Error bars are for a binomial distribution.

The extent to which developers regard `->` as an operator that can have a non-trivial runtime overhead, compared to other object access expressions is not known.

Table .2: The percentage of tested-expressions containing the given number of operators listed selection statement (all binary operators and unary excluding **`sizeof`** and casts). The second number in each column includes the `->` operator in the total.

Selection statement	0	1	2
switch	89.3/59.7	8.7/35.1	1.9/4.5
if-else-if	89.8/67.3	9.3/29.9	0.9/2.7
if-if	85.9/70.8	13.5/26.2	0.5/3.0

It is tempting to observe that the number of operators in switch-statement and if-else-if sequence controlling expressions is very similar when the `->` operator is excluded, but that when this operator is counted the most similar pairing is if-else-if and if-if. However, without a model of behavior it is not possible to say anything statistically significant.

4.4 Number of equality tests controlling conditional arms

A controlling expression may map to multiple case-labels. For instance, each equality test in an expression containing one or more logical-OR operators is mapped to a separate case-label and a *between operation* implemented using a logical-AND operator and relational operators to compare against values within a range is mapped to the corresponding range of case-labels.

Take as an example the controlling expressions present in an if-if sequence. If each controlling expression is independent of the others, then the probability of two equality tests, for instance, occurring in any of these expressions is constant and thus given a large sample the distribution of two equality tests has a binomial distribution. The same argument can be applied to other numbers of equality tests and other kinds of sequence.

For each measurement point in Figure .3 the associated error bars span the square-root of the variance of that point (assuming a binomial distribution, for a normal distribution the length of this span is known as the standard deviation). The error bars overlap suggesting that the apparent difference in percentage of equality tests in each kind of sequence is not statistically significant.

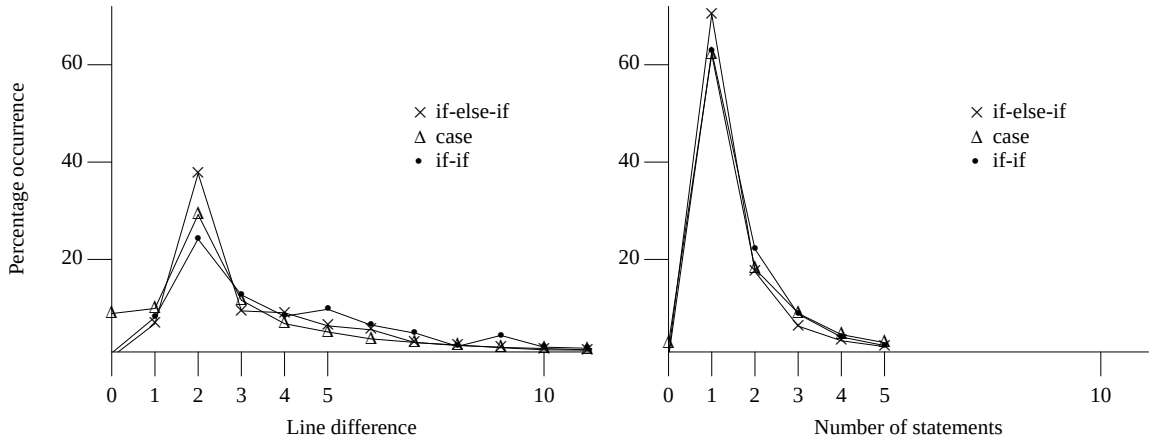

Figure .4: Two methods of measuring the *length* of a conditionally executed arm are plotted. The left plot is based on the difference in visible source line number between consecutive **`if`** tokens in an if-else-if and if-if sequence, and between a **`case`** token and the **`break`**, **`return`** or **`continue`** that terminates that arm. The right plot is based on the number of *top-level* statements, that is statements contained within any nested block are not counted. Sequences containing more than five statements were not counted, resulting an overestimation of the actual percentage of shorter sequences.

4.5 Number of statements in conditional arm

How much code appears in the conditional arms of if-else-if, if-if sequences and case-labeled statement sequences?

Most of the Coccinelle patterns output line number information. This information can be used to calculate the difference in visible source lines between adjacent if-statement controlling expressions and between the last **`case`** labeling a given statement sequence and the final statement that caused the flow of control to leave the switch-statement. Those line number difference measurements appears in the left plot of Figure .4.

An example of line number difference is provided by the first if-else-if code fragment given in the introduction, where the difference between controlling expressions is 2; the distances in the first switch-statement example are both 1.

Because of the variety of different ways in which if-else-if, if-if and case-labeled statements can be visually laid out, line number differences may not be a consistent measure of the quantity of code present in the various language constructs.

The right plot of Figure .4 is based on counting statements. The Coccinelle patterns used did not count statements within nested blocks. This means, for instance, that a for-loop was counted as a single statement and any statements nested within its associated block did not contribute towards the total statement count.

The patterns used counted code containing a maximum of five statements and consumed over a day of cpu time; it was felt that measuring more statements was unlikely to change the pattern of usage seen in Figure .4. It is estimated that had longer sequences been counted the actual percentage of shorter sequences would have been around 20% lower..

Within a case-labeled statement sequence any terminating **`break`** statement is not included in the total statement count for that sequence. However, if the sequence is terminated by a **`return`** statement this statement is counted towards the total statement count for the sequence.

The two plots in Figure .4 suggest that if-else-if, if-if and case-labeled arms are very similar both in the visual number of lines and the number of top-level statements they contain.

5 Discussion

Some of the measured characteristics where a notable difference was seen between **`if`** and **`switch`** statement usage include:

- Figure .2 suggests the number of conditional arms in the construct and/or the number of controlling expressions/case-labeled statement sequences have a large effect on the likelihood of a particular kind of selection statement being used. It is not possible to separate out the relative contributions of the number of controlling expressions and number of conditionally executed arms with the data available.
- the use of function calls in the tested-expression, see Table .1.

The characteristics that appear to have the largest effect on selection statement usage are the number of conditional arms in the construct and/or the number of controlling expressions/case-labeled statement sequences. The second part of this paper describes an experiment that asked subjects to write code based on specifications that involved different numbers of control expressions.

6 Further reading

A good introduction, at an undergraduate level, to the various algorithms people are thought to use when making decisions is provided by: "The Adaptive Decision Maker" by John W. Payne, James R. Bettman and Eric J. Bettman, published by Cambridge University Press; ISBN 0-521-42526-3.

7 Acknowledgements

The author wishes to thank everybody who volunteered their time to take part in the experiments and ACCU for making a slot available, in which to run the experiment.
Thanks to Julia Lawall for suggestions for improving the Coccinelle patterns used and responding very promptly to bug reports; also thank to Yoann Padioleau for support using Coccinelle.

References

1. G. Gigerenzer, P. M. Todd, and The ABC Research Group. *Simple Heuristics That Make Us Smart*. Oxford University Press, 1999.
2. D. M. Jones. Who guards the guardians? `www.knosof.co.uk/whoguard.html`, 1992.
3. D. M. Jones. The new C Standard: An economic and cultural commentary. Knowledge Software, Ltd, 2005.
4. Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 3^{rd} ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 247–260, Mar. 2008.
5. G.-R. Uh and D. B. Whalley. Effectively exploiting indirect jumps. *Software–Practice and Experience*, 29(12):1061–1101, Oct. 1999.