



Coding guidelines for static analysis

25/07/2009

Université Paul Sabatier de Toulouse

Armelle Bonenfant, Hugues Cassé,
Marianne De Michiel, Christine
Rochange, Pascal Sainrat

Table of contents

Object of this report.....	4
A few words on oRange	4
A few words on Ottawa	4
Organization	4
Table of contents.....	2
oRange Coding Rules.....	5
1.1. Control statement expressions.....	5
1.1.1. No use of assignment operators in expressions that yield a Boolean Value	5
1.1.2. Expressions of a <code>for</code> statement reserved only for loop control	5
1.1.3. Loop counter not modified in statements conditioned by an <code>if</code>	6
1.1.4. Bounds depending on inputs.....	6
1.2. Control flow	7
1.2.1. No <code>goto</code>	7
1.2.2. No <code>break</code> , <code>continue</code> , <code>return</code>	7
1.2.3. Switches should be well-structured.....	7
1.3. Functions	7
1.3.1. No variable number of arguments.....	7
1.3.2. No recursivity.....	7
1.3.3. Pointer as a parameter of a function	7
1.4. Pointers.....	8
1.4.1. Pointer arithmetic only for arrays	8
1.4.2. Levels of pointer indirection.....	8
1.5. Enum and Union	8
Ottawa Coding Rules.....	8
1.6. Non supported statements	8
1.6.1. Switches	8
1.6.2. <code>setjmp/longjmp</code>	8
1.6.3. Functions which modify the stack pointer	9
1.6.4. System calls.....	9
1.6.5. Avoid hardware control instructions.....	9
1.7. Statements increasing overestimation.....	9
1.7.1. Dynamic memory usage	9
1.7.2. Dynamic size array allocated in stack	10
1.7.3. <code>alloca()</code> primitive.....	10
1.7.4. Function with variable number of arguments	10
1.7.5. Avoid unbalanced selection in loops	10
1.8. Ways of decreasing overestimation.....	11

1.8.1.	Multiple execution paths	11
1.9.	Decreasing overestimation by helping the analyzer	12
1.9.1.	Imprecise loop bounds	12
1.9.2.	Infeasible paths identification	13

Object of this report

The aim of this report is to provide coding guidelines to Merasa partners in order to produce code that has good properties for static analysis by the oRange and Ottawa tool. Most of these guidelines concern C source code and some of them are for assembly code only.

This report is not a report on general good rules of coding as, for example, [1]. Note that several of our rules are also in [1] so if you are used to respect the Misra rules then it is highly probable that your code will be correctly analyzed with oRange and Ottawa.

A few words on oRange

oRange is a tool whose aim is to calculate automatically bounds of loops. Analysis is made on the source code. It starts by rewriting the source code in order to produce something analyzable. Then, an analysis of this rewritten code is made in order to define loop bounds. It is not able to analyze all kind of loops and structures. It may find exact bounds or overestimated bounds depending on the complexity of the expressions.

A few words on Ottawa

OTAWA is a C++ framework dedicated to the estimation of Worst-Case Execution Times using static analysis techniques. It includes a number of facilities to handle binary code and has been designed to support different architectures. It constitutes an efficient framework to develop analysis modules that, when applied in sequence on a binary code, help in estimating its WCET tightly. It supports (or will support) everything needed for the MerasaCore architecture. Ottawa receives loop bounds produced by oRange and, if oRange cannot provide bounds, it is possible to give loop annotations in order to tighten the analysis. Static analysis techniques that are in Ottawa overestimates more or less the WCET depending on the structure of the object code and in some cases is not able to calculate a WCET.

A very simple example case where Ottawa is not able to calculate a WCET is:

```
while (1)
{
    statements ;
}
```

The WCET of this little program cannot be computed as it is a non sense since the program runs forever. The user should provide an annotation on the while loop...

Organization

Coding rules for oRange may be neglected if annotations are used to provide flow information to Ottawa. However, it's better to leave oRange find the loop bounds as it is less error-prone.

On the contrary, most Ottawa rules should be respected, except where specified when an annotation can be used instead.

Thus, the coding rules are given first for oRange then for Ottawa.

oRange Coding Rules

All oRange coding rules concern the control flow of the program. Some rules must absolutely be respected because orange is not able to deal with them. In the worst case, not respecting them may conduct oRange to be in the impossibility of finding some loop bounds and in the best case, to produce overestimated loop bounds. This is mentioned for each rule.

Control statement expressions

1.1.1. No use of assignment operators in expressions that yield a Boolean Value

An expression which is considered to produce a Boolean value should not contain an assignment.

Example

Avoid	Prefer
<pre>if ((x = y) != 0) { foo() ; }</pre>	<pre>x = y ; if (x != 0) { foo() ; }</pre>

Comment

This is not mandatory but eases the analysis. Not respecting it may prevent oRange to find some loop bounds.

1.1.2. Expressions of a for statement reserved only for loop control

The three expressions of a for statement should be as simple as possible and concern only the loop counter.

Example

Avoid	Prefer
<pre> flag = 1 ; for (i = 0 ; (i < 5) && (flag == 1) ; i++) { ... ; if (...) { flag = 0 ; } } </pre>	<pre> flag = 1 ; for (i = 0 ; (i < 5) && (flag == 1) ; i++) { ... ; if (...) { break ; } } </pre>

Comment

Orange might be able to produce a loop bound in the case of an && operator but will, in many cases, produce the highest, 5 in the example. Any other operator is not supported.

1.1.3. Loop counter not modified in statements conditioned by an if

This rule can be restricted to the case where the loop counter is modified in the opposite direction than in the `for` statement.

Example

Avoid	OK
<pre> for (i = 0 ; (i < 5) ; i++) { ... ; if (...) { i-- ; } } </pre>	<pre> for (i = 0 ; (i < 5) ; i++) { ... ; if (...) { i++ ; } } </pre>

Comment

The left example should be absolutely avoided. It's better, if possible to avoid the right example.

1.1.4. Bounds depending on inputs

Avoid as much as possible bounds depending on inputs. The worst-case will be the size of the input. In many cases, a loop depending on an input can be replaced by a loop with a tighter bound. In some cases, for example when going through an array, it is possible to proceed by dichotomy which gives a lower bound.

Control flow

1.1.5. No goto

This is not mandatory but avoiding `goto` eases analysis.

1.1.6. No break, continue, return

In some cases, the use of `break`, `continue` or `return` in the middle of a control flow structure may avoid `oRange` to calculate a loop bound.

1.1.7. Switches should be well-structured

Keep switches regular, that is, keep cases and breaks at the same nesting level.

Example

Avoid	Prefer
<pre>switch (v) { case 1 : case 2 : if (...) { ... ; break; } ... break ; }</pre>	<pre>switch (v) { case 1 : case 2 : if (...) { ... } else { ... } break ; }</pre>

Functions

1.1.8. No variable number of arguments

This precludes the use of `va_arg`, `va_start`, `va_end`.
This is not supported.

1.1.9. No recursivity

Generally speaking, recursivity should be avoided because it carries with it the danger of exceeding available stack space.

`oRange` supports some simple forms of recursivity (cases where the recursivity can be transformed in a `for` loop).

1.1.10. Pointer as a parameter of a function

If a pointer is a parameter of a function and the pointer is not modified in the function, it is better to use `const`.

Example

Code	OK if
<code>foo(int *a, const int *b)</code>	a is modified in the function. b is not.

Pointers

1.1.11. Pointer arithmetic only for arrays

Arithmetic on pointers should only be used for array indexing.

1.1.12. Levels of pointer indirection

As the levels of pointer indirection increases, difficulty of analysis increases. Yet, we can ensure only one level.

Pointers to functions are not supported.

Example

Allowed	Not allowed
<code>*p</code>	<code>**p /* and higher ! */</code>

Enum and Union

Enums are supported. Unions are not supported.

Ottawa Coding Rules

Non supported statements

1.1.13. Switches

Large switches are not supported because they are translated by the compiler as branch tables. A series of if ... then...else is preferable.

Keeping a switch is possible but it will need manual annotations which is error prone. Notice that some compilers provide options to configure the way the switch is translated and to prevent the generation of indirect branch tables. OTAWA provides for some architectures and some compilers (but not the Tricore) switch analysis through pattern recognition.

1.1.14. setjmp/longjmp

setjmp and longjmp are not supported.

1.1.15. Functions which modify the stack pointer

This is only possible in assembler and not in C. It is forbidden to have a function which modifies the stack pointer so that the pointer is not the same after the function has been executed.

1.1.16. System calls

System calls are not supported as the target depends on the configuration of the underlying OS. Yet, the actual target of the system call may be provided by hand and considered as a normal branch provided the system routine is in the binary code.

1.1.17. Avoid hardware control instructions

Most of these instructions (controlling the behavior of caches, MMU, etc) are not handled correctly by the WCET analysis. Do not expect their effect to be correctly handled unless it is explicitly stated in documentation. For example, if you use an instruction which inhibits the cache, the fact that the cache is inhibited for the rest of the program is not taken into account.

Statements increasing overestimation

1.1.18. Dynamic memory usage

The management of dynamic memory involves complex algorithms that make harder the prediction of the used memory. It may even be impossible to predict it if the program contains complex allocation patterns. This has mainly a bad impact on the prediction of the data cache use (for example, all accesses will be considered as misses).

Instead of using standard allocation primitives of C, a good solution is to develop specialized allocators based on array whose memory area is well known because it is reserved statically.

Example

Avoid	Prefer
<pre> struct node_t { struct node_t *next; ... } *p = NULL, *q; for (...) { q = (struct node_t *)malloc(sizeof(struct node_t)); q->next = p; p = q; ... } </pre>	<pre> struct node_t { struct node_t *next; ... } *p = NULL, *q, tab[MAX]; int next_block = 0; for (...) { q = &tab[next_block]; next_block++; q->next = p; p = q; ... } </pre>

1.1.19. Dynamic size array allocated in stack

Some compiler allows declaring arrays with dynamic size in the local variables of a function. This makes the address in the stack dependent on the flow of the data and makes harder the prediction of the data cache usage.

In addition, the kind of allocation makes harder or impossible the prediction of the stack size if the data flow determining the size is dependent on the program input.

Example

Avoid	Prefer
<pre>void f(..., int n, ...) { int t[n]; ... }:</pre>	<pre>int t[MAX_N]; void f(..., int n, ...) { ... }</pre>

1.1.20. `alloca()` primitive

The use of this standard C primitive is not advised by the man pages. In WCET computation, it produces the same effect as described in the previous section.

1.1.21. Function with variable number of arguments

The use of function with a variable number of arguments is not advised but it should induces only very few loss of determinism.

1.1.22. Avoid unbalanced selection in loops

When a condition is not simple enough to be evaluated by the WCET analyzer, the most costly part of the selection will be considered by the WCET analysis. If the difference of times between the selection alternatives is big, it will induces a big overestimation. This fact is especially true when the selection has an empty “else” part and the “then” part is used only once in a loop as shown in the following examples.

Example

Avoid	Prefer
<pre>for (i = 0; i < N; i++) { if (i == 0) { /* initialisations */ } else { ... } }</pre>	<pre>i = 0; /* initializations */ for(i = 1; i < N: i++) { ... }</pre>

Comment

It is a common practice to perform some special processing like initializations at first iteration and share the same processing for other iterations.

Example

Avoid	Prefer
<pre> for(i = 0; i < N; i++) { if(... t[i] ...) { /* work on t[i] */ break; } ... } </pre>	<pre> int found = 0; for(i = 0; i < N; i++) { if(... t[i] ...) { found = 1; break; } ... } if(found) { /* work on t[i] */ } </pre>

Comment

The rationale behind this example is that the algorithm is looking for a unique $t[i]$ in order to perform a processing on it. If the condition is too complex, the WCET analyzer will conclude that the work is done for each element of t in the left version of the program and only once in the right one

Note

The inserted boolean `found` does not break compatibility with `oRange` as it is not used in the loop condition.

Decreasing overestimation by avoiding multiple execution paths

The main cause of overestimation is due to the lack of determinism in the application. From a C source point of view, indeterminism is caused by the number of execution paths. In the analysis, every join of paths may cause a loss of precision. To prevent from having too much paths, one has to avoid as much as possible the use of

- conditional statements,
- loops with a variable bound.

Compiler's optimizations provide ways to enforce these statements. The algorithms below have been designed to improve performance by exploiting parallelism in the processor but, as a side effect, they tend to reduce the number of paths by aggregating adjacent code blocks. The price is an increase in the size of the program due to code duplication. As we are targeting embedded real-time application, a trade-off between determinism and code size should be found. These optimizations can be done manually on the source code because using those of the compiler may lead to not analyzable code.

These optimizations are:

- function inlining – a small function call is replaced by its code,
- loop fusion – adjacent loops with the same number of iterations are merged,

- loop unrolling – the loop is unrolled to exhibit more parallelism in each iteration,
- single-path code – use of guarded instructions to avoid branches.
- super-block – if a selection is followed by a simple block, this block may be duplicated in each branch.

Decreasing overestimation by helping the analyzer

In some cases, it is not possible to rewrite the code in order to avoid overestimation caused by the flow analysis. For some of them, one can provide manual annotations in order to help Ottawa to reduce the overestimation. Two examples are given below.

1.1.23. Imprecise loop bounds

Loops with complex or variable bounds (most often these are nested loops) are a source of overestimation because, for each entry in the loop (i.e. each time the outer loop is executed) the maximum number of iterations might always be considered.

In order to obtain a more precise loop behaviour description, one should give several annotations on the inner loop:

- total: the maximum number of iterations over all the program execution,
- minimum: minimum number of iterations,
- context: different maxima, total or minima according to the call chain leading to the loop.

Example

Sample	Annotations
(1) <code>for(i = 0; i < N; i++)</code>	Loop (1)
(2) <code>for(j = 0; j < i; j++) {</code>	maximum = N
<code>/* body */</code>	total = N
<code>}</code>	Loop (2)
<code>}</code>	minimum = 0
	maximum = N
	total = N * (N + 1) / 2

Comment

The inner loop iterates at most N times each time it is called and as loop (1) iterates N times, the analyzer may conclude that loop (2) iterates N^2 in total. But, the programmer knows that the inner loop iterates $N(N + 1) / 2$.

Example

Sample	Annotations
<pre>void f(..., int n, ...) { int i; for(i = 0; i < n; i++) { ... } }</pre>	<p>Loop in f maximum = 100 total = 105</p> <p>Call (1) / loop maximum = 5 total = 5</p>
<pre>int main() { ... (1) f(..., 5, ...); ... (2) f(..., 100, ...); ... }</pre>	<p>Call (2) / loop maximum = 100 total = 100</p>

Comment

Here, one can see that the overall maximum iteration is 105 while, in case (1), the local maximum is 5 and in case (2), it is 100. Both local maxima are very different and the overall maximum would induce a big overestimation. Considering only the total is also not enough as the computation may be free to assign 100 iterations to the first call to maximize other features cost like cache misses. In fact, it is more precise to consider maxima and total for each call context.

1.1.24. **Infeasible paths identification**

An important source of overestimation is the inclusion in the WCET analysis of infeasible paths, that is, paths that are not in the set of execution paths due to the program semantics. Although WCET analyzers can find automatically some of these paths, it remains some constructions that remain too hard to handle but the developer may help to avoid such a kind of overestimation.

Example

Sample	Annotations
<pre>for(i = 0; i < 100; i++){ if(i % 2 == 0) { (1) /* light block */ } else { (2) /* heavy block */ } }</pre>	<p>Annotation on (2) execute at most 50 times</p>

Comment

Of course, in this very simple example, you can also provide an annotation on (1)! Yet, it may be hard or error prone to find by hand all infeasible paths. A better approach is

to consider only cases of a selection where the “then” and “else” parts have very different costs for the WCET computation and to put an annotation on the most costly branch.

Bibliographie

[1] Mira, limited., *Misra-C: Guidelines for the use of the C language in critical systems*. 2004. 978-0-9524156.