Extending Clang for checking compliance with automotive coding standards

Syrmia LLC
Overview of the talk

Autosar, problem analysis and objectives
  Autosar guidelines for C++14 language
  Checking compliance with automotive coding standards

Clang’s support and interfaces
  Support within Clang
  Interfaces for semantic analyses
  Sophisticated static analysis

AutoCheck
  Implementation details
  Results
  Comparison to Clang-Tidy

Conclusions and Further Work
Autosar guidelines for C++14 language

- Autosar guidelines are tailored to improve security, safety and quality of software in critical and safety-related systems (primarily automotive, but these guidelines can be used in other embedded application sectors)

- 402 rules:
  - \( \sim 200 \) derived/based on the existing C++ standards
  - \( \sim 150 \) adopted without modifications from MISRA C++:2008 (64% of MISRA is adopted without modifications)
  - \( \sim 60 \) based on research papers, other literature or other resources
Autosar guidelines for C++14 language

Rule classification according to

- **Obligation level**: required and advisory
- **Allocated target**: implementation, verification, toolchain and infrastructure
- **Enforcement by static code analysis tools**
  - **Automated**: rules that are automatically enforceable by means of static analysis.
  - **Partially automated**: rules that can be supported by static code analysis, e.g. by heuristic or by covering some error scenarios (as a support for a manual code review)
  - **Non-automated**: rules where the static analysis cannot provide any reasonable support
Autosar guidelines for C++14 language

Our focus: \(\sim 340\) rules

- **Implementation** based rules
- Rules that can be **automated**
- Rules that are **required or advisory**
Examples

• Simple decidable rules:
  • Trigraphs shall not be used (-Wtrigraphs)
  • Literal suffixes shall be upper case.

• Decidable rules:
  • Different identifiers shall be typographically unambiguous
  • The continue statement shall only be used within a well-formed for loop.

• Undecidable rules (run-time features):
  • A project shall not contain unreachable code (-Wunreachable-code).
  • The right hand operand of the integer division or remainder operators shall not be equal to zero (-Wdivision-by-zero).
Problem analysis

- Big number of rules (∼ 340)
- Big differences between rules: some are easy to check while some are very complex
- False alarms vs undiscovered violations
- Existing support:
  - Clang,
  - Clang’s AST Visitors and AST Matchers,
  - Clang-tidy, as a framework for using AST Matchers,
  - Clang Static Analyzer
Objectives

- No undiscovered violations
- Efficient and precise analysis
- User friendly: like compiler warnings, but with additional control over reporting mechanism
- Good design principles: easy to maintain and verify
Existing support within Clang

- 44 rules that are supported or partially supported by Clang:

  Examples:

  - Supported:
    - Trigraphs shall not be used (-Wtrigraphs).
  - Partially supported:
    - The form of delete operator shall match the form of new operator used to allocate the memory (-Wmismatched-new-delete).
    - The right hand operand of the integer division or remainder operators shall not be equal to zero (-Wdivision-by-zero).
Improvements of Clang’s diagnostics

- It is possible to directly improve Clang's diagnostics by adding support for some simple checks when appropriate.
- Definition of appropriate: whenever that does not affect Clang’s efficiency and whenever it is easy to maintain the extended code between different versions of Clang.
- We keep Clang’s behavior unchanged, unless our flags are present.
Semantic analyses via AST Visitors and AST Matchers

- Two interfaces for semantic analysis:
  - AST Matchers — provide a simple, powerful, and concise way to describe specific patterns in the AST.
  - AST Visitors — provide using the full power of the Clang AST

- Pros and cons: matchers should be easier to implement and maintain, but do not always give you a full control over the AST, Clang-Tidy gives a valuable framework for writing code-style checks by AST Matchers, efficiency issues

- Experimental analysis
Example:
A8-4-1 Functions shall not be defined using the ellipsis notation.

```c
void function1(int a, ...) {
    // ...
}
```

AST:

```
'-FunctionDecl 0x12223e8 <48.cpp:18:1, col:29> col:6 function1 'void (int, ...)'  
|'-ParmVarDecl 0x1222310 <col:16, col:20> col:20 a 'int'  
  '-CompoundStmt 0x12224d8 <col:28, col:29>  
    ...
```
Matchers are easier to implement and maintain

Example:

A8-4-1 Functions shall not be defined using the ellipsis notation.

Visitor:

```cpp
bool VisitFunctionDecl(const FunctionDecl *FD) {
    if (FD->isVariadic()) {
        // report warning
    }
    return true;
}
```

Matcher:

```cpp
functionDecl(isVariadic())
```
AST Visitors vs AST Matchers

Example:

Rule 6–6–5 A function shall have a single point of exit at the end of the function.

Visitor:

```c
bool VisitReturnStmt(const ReturnStmt *RS) {
    ++returnCount;
    if (returnCount > 1) { /*report warning*/ }
    return true;
}
```

Matcher:

```c
functionDecl(hasDescendant(returnStmt().bind("return")),
             hasDescendant(returnStmt(unless(equalsBoundNode("return")))));
```
AST Visitors vs AST Matchers

• Counting becomes tiresome if we count for more than just two
• In addition, matchers do not naturally solve the problems concerning the order of statements that is important in some rules (like in: The goto statement shall jump to a label declared later in the same function body), especially if that is important as a part of some sub-goal within the rule
• There are also additional examples when Matchers are not the first choice
Experimental setup for measuring efficiency

- Write several AST Matchers and AST Visitors checking the same property
- Generate code that
  - Contains only the expected structure that is checked
  - Does not contain any of the expected structure that is checked
  - Contains approximately 5% of code with the expected structure
- Vary size of the generated code: 100, 500, 1000, 2000, 5000, 10000 LOC
- Measure 100 times and take the average
Experimental setup

- Measure the efficiency also on *Automotive Grade Linux open source code*, which serves as an industry standard to enable rapid development of new features and technologies.
- AGL contains a code base with many sub-projects and we use several sub-projects as testing benchmarks.
Results

- No big differences between different sizes of code and between different checks
- The smallest difference — no expected structure that is checked:
  - Visitors are as fast as matchers, i.e. there are no big differences
- The biggest difference — only the expected structure that is checked
  - Visitors are faster compared to matchers between 3.1 and 5.1 times
- On code with 5 percent of expected structure
  - Visitors are faster compared to matchers between 1.2 and 1.5 times
- On AGL code
  - Visitors are faster compared to matchers between 2 and 3 times
Static Analyzer

- Source code analysis tool for bug finding
- Takes into account CFG, not only AST
- Based on bounded model checking and considers loops with just a few loop unrollings, and therefore should not report false positive results but can have false negatives
- Much slower than compilation (visitors or matchers)
AutoCheck

- Implemented 190 rules from Autosar C++14 guidelines
  - Some of these rules are language independent or can be used on C code as well (∼120 rules)
- Some rules are implemented directly within Clang (∼80 rules), others are implemented through AST Visitors
  - Visitors are grouped into clusters that maximize efficiency
- Four rules are additionally supported by more precise analysis through Static Analyzer (division by zero, null pointer dereferencing, pointer arithmetic, recursive function calls)
- Autocheck uses llvm’s infrastructure for testing (each rule is covered with several positive/negative test cases), and also AGL code
Usage

- AutoCheck is used internally on projects that require compliance with Autosar guidelines
- The obtained feedback is used for guiding the development of the tool
- AutoCheck is an extension of Clang so plugins for Clang’s integration within different software development environments can be used
Controlling the output

- New options that differ to standard compiler options
  - Limit the number of warnings issued for each violated rule and stop performing the analysis for each rule after its limit is reached:
    option -autocheck-limit=N
  - Analyze and report warnings only between some specific lines
    -autocheck-between-lines=<from-line>,<to-line>
- Suppress warnings corresponding to macro extensions
  -autocheck-dont-check-macro-expansions
- Disable checks within headers
  -autocheck-dont-check-headers
Automotive Grade Linux open source code

- The efficiency of AutoCheck is measured on different corpora.
- When building AGL subprojects:
  - If only options that are implemented directly within Clang are included, time that AutoCheck takes is bigger between 1.1 and 1.7 times (compared to Clang).
  - If all visitors are also included, time that AutoCheck takes is bigger between 1.7 and 9.2 times (compared to Clang).
- These differences depend on number of violated rules and on number of times the rule is violated.
Automotive Grade Linux open source code

• Options `-autocheck-limit` and `-autocheck-dont-check-headers` reduce significantly these time differences

• Examples:
  • In `qrc_hvac.cpp`, there are **11** different rules that are violated
    ~ **15K** times (headers included),
    ~ **3K** times (headers not included)
  • In `qrc_images.cpp`, there are **11** different rules that are violated
    ~ **97K** times (headers included),
    ~ **23K** times (headers not included)
Clang’s code base

- There are 129 rules violated within Clang’s code base
  - 8 rules are violated less than 10 times
  - 11 rules are violated between 10 and 100 times
  - 9 rules are violated between 100 and 1,000 times
  - 25 rules are violated between 1,000 and 10,000 times
  - 37 rules are violated between 10,000 and 100,000 times
  - 39 rules are violated more than 100,000 times

- The biggest number of warnings
  fixed width integer types from `<cstdint>`, indicating the size and signedness, shall be used in place of the basic numerical types
Comparison to Clang-Tidy

- **Clang-Tidy**
  - is a C++ "linter" tool, support for different coding conventions and an interface for adding new checks
  - is LibTooling-based tool, uses AST Matchers
  - can run Static analyzer

- **AutoCheck**
  - support for C++14 Autosar guidelines, custom tailored solution
  - can be invoked as a Clang option, is based on Clang and AST Visitors
  - can run Static analyzer
Conclusions and Further work

- LLVM/Clang give several frameworks for implementing syntax and semantic analysis
- We had many different decisions to make on our road, that were explained and commented during this talk
- We successfully implemented 190 rules from Autosar guidelines, together with different options controlling the output in the user friendly way
- Further work: implement the rest of the rules
Contact us

SYRMIA
Belgrade Office Park
Đorda Stanojevića 12
Belgrade 11070
Serbia