A «NEW» STATIC ANALYZER: THE COMPILER

Maurizio Martignano
Spazio IT – Soluzioni Informatiche s.a.s
Via Manzoni 40
46030 San Giorgio Bigarello, Mantova
https://www.spazioit.com

June 2019
Agenda

- Need for Speed
- Libraries, Libraries and again Libraries
- Clang/LLVM – SonarQube
- SAFe Toolset
- Future Activities
Need for Speed

June 2019
Need for Speed

- The size of software codebases is increasing dramatically:

<table>
<thead>
<tr>
<th>Year</th>
<th>System</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1974</td>
<td>F16A Plane</td>
<td>135 K</td>
</tr>
<tr>
<td>1981</td>
<td>Space Shuttle PFS</td>
<td>400 K</td>
</tr>
<tr>
<td>2008</td>
<td>ESA ATV</td>
<td>1 M</td>
</tr>
<tr>
<td>2012</td>
<td>NASA Curiosity</td>
<td>2.5 M</td>
</tr>
<tr>
<td>2012</td>
<td>F35 Plane</td>
<td>10 M</td>
</tr>
<tr>
<td>Nowadays</td>
<td>Car</td>
<td>10-150 M</td>
</tr>
</tbody>
</table>

- Compilers and Static Analyzers need to be fast and efficient (i.e. able to “digest” large codebases in a reasonable time).
Need for Speed

- Deep vs. Shallow Parsing

- Unforgiving vs. Forgiving Parsing
Are we siblings?

I don’t know! Do you use my libraries?
Suppose that for a given language we have a compiler and a static analyzer that are two separate software products, using different libraries and technologies (each one of them as its own lexer, parser, semantic analyzer and so on).

Suppose the developer community behind that language and tools is not very big and doesn’t have many resources, lots of energy.

In case the language changes, evolves, for whatever reason, which of the two tools (the compiler or the static analyzer) will keep up with the language evolution?

In the same way, which of the two tools will be more performant?
- PC-Lint does not support the latest C/C++ Standards.
- Frama-C Semantic Analyzer cannot process all C/C++ constructs.
- Ada ASIS does not support Ada 2012 (but the GNAT compiler does).
- In the Ada “libadalang” website we have: “Libadalang does not (at the moment) provide full legality checks for the Ada language. If you want such a functionality, you’ll need to use a full Ada compiler, such as GNAT.”
- and so on…
“The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. (...) The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many CPUs. (...) Clang is an LLVM native C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles.”

In fewer words Clang/LLVM is a compilation toolchain where absolutely everything is built in a modular fashion as collection of reusable libraries.
In the Clang/LLVM toolchain the two static analyzers are Clang-Check (a.k.a. Clang-SA) and Clang-Tidy.

Clang-Check relies on a set of Clang modules to perform things like lexical analysis, parsing, semantic analysis, AST manipulation and the like.

Clang-Tidy relies on the very same Clang modules plus some additional modules of Clang-Check itself (this is why Clang-Tidy can be considered a sort of superset of Clang-Check).
Libraries, Libraries and Again Libraries

```c
CXCursor cursor = clang_getTranslationUnitCursor(unit);
clang_visitChildren(
    cursor,
    [](CXCursor c, CXCursor parent, CXClientData client_data)
    {
        if (clang_Location_TurnMainFile (clang_getCursorLocation (c))) {
            if (strcmp("Function", clang_getCursorKindSpelling (c)))
                printf ("Function '%s' not found in file " CLANG_FORMAT_STRING("%s") ",
                    clang_getCString(clang_getCString(clang_getCursorKindSpelling (c))),
                    
                    return CXChildVisitEnd;
                }
        
        return CXChildVisitEnd;
    },
    nullptr);
clang_disposeTranslationUnit(unit);
```
“libclang” is nothing but a simple C API (with Python bindings) exposing Clang functionalities (i.e. modules) to external applications (deep / forgiving parsing);

thanks to “libclang” also these third-party applications can use the very same modules/libraries of Clang (for instance they could parse a C program as efficiently as Clang does).
Libraries, Libraries and Again Libraries

June 2019
Clang / LLVM – SonarQube Integration

June 2019

© 2019 Spazio IT - Soluzioni Informatiche s.a.s.
SonarQube – What is it?

Source Code Files → SonarQube Engine → SonarQube Database → Analyses Results

Source Code Files

Analyses Results

SonarQube Database

SonarQube

Database

Engine

Analyses

Results
SonarQube / Plugins / Sensors

SonarQube

Plugin-1
- e.g. Ada

Plugin-I
- e.g. C/C++

Plugin-M
- e.g. Java

Pre-Processing
- e.g. scanning and parsing
  - Sensor-1
    - e.g. CppCheck
  - Sensor-J
    - e.g. PC-Lint
  - Sensor-M
    - e.g. GCOV

Post-Processing
- e.g. MeasureComputers (Ex. Decorators)
SonarQube C++ plugin (Community)

- Parser supporting C89, C99, C11, C++03, C++11, C++14 and C++17 standards
  - Microsoft extensions: C++/CLI, Attributed ATL
  - GNU extensions
  - CUDA extensions

- Sensors for static code analysis:
  - Cppcheck warnings support (http://cppcheck.sourceforge.net/)
  - GCC/G++ warnings support (https://gcc.gnu.org/)
  - Clang Static Analyzer support (https://clang-analyzer.llvm.org/)
  - Clang Tidy warnings support (http://clang.llvm.org/extra/clang-tidy/)
  - PC-Lint warnings support (http://www.gimpel.com/)
  - (…) many others

June 2019
Clang / LLVM – SonarQube Integration
Clang 9 documentation

JSON Compilation Database Format Specification

This document describes a format for specifying how to replay single compilations independently of the build system.

Background

Tools based on the C++ Abstract Syntax Tree need full information how to parse a translation unit. Usually this information is implicitly available in the build system, but running tools as part of the build system is not necessarily the best solution:

- Build systems are inherently change driven, so running multiple tools over the same code base without changing the code does not fit into the architecture of many build systems.
- Figuring out whether things have changed is often an IO bound process; this makes it hard to build low latency en-usertools based on the build system.
- Build systems are inherently sequential in the build graph, for example due to generated source code. While tools that run independently of the build still need the generated source code to exist, running tools multiple times over unchanging source does not require serialization of the runs according to the build dependency graph.
SAFe Toolset

compile_commands.json

Project.json

normalized compile_commands.json
Project.Int run_pclint.sh
Project.cppcheck run_cppcheck.sh
Project_clang-sa.sh
Project_clang-tidy.sh
Project_comp.sh
sonar-project.properties run_sonar.sh
The **SAFe Toolset** is an **Ubuntu Virtual Machine** containing various open source tools that can be used to perform **Software Verification** and **Validation**.

In particular the current version (June 2019) of the SAFe VM contains:

- **Clang** – v. 9.0.0 - [https://clang.llvm.org](https://clang.llvm.org) – the “new” compiler toolset from LLVM Foundation, with its **Clang-SA** and **Clang-Tidy** static analyzers.
- **SonarQube** – v. 7.7. – [https://www.sonarqube.org/](https://www.sonarqube.org/) - a code quality platform used to show and manage the issues found by the static analyzers.
Optionally the SAFe VM may also contain:
- **PC-Lint** (or PC-Lint Plus) – v. 9.0.0L - [https://www.gimpel.com/](https://www.gimpel.com/) - but its license needs to be acquired from Gimpel.

Apart from the static analyzers the SAFe VM contains also some (native and cross) build environments, that is:
- **GNU GCC** Version 7.3.0 - [https://gcc.gnu.org/gcc-7/](https://gcc.gnu.org/gcc-7/) - Native
- **Clang** Version 9.0.0 - [https://clang.llvm.org](https://clang.llvm.org) - Native and Cross (Multiplatforms – use the command “llc --version” to see the supported architectures).
- **GNU Arm Embedded Toolchain** - v. 5-2016-q3 - [https://launchpad.net/gcc-arm-embedded](https://launchpad.net/gcc-arm-embedded) - Cross.
Should a user need to work on a codebase not supported by the provided build environments, she would need to install the corresponding compilation toolchain.

Additionally Spazio IT has complemented the SAFe Toolset with:

- a specially modified version of SonarQube - [https://www.sonarqube.org/](https://www.sonarqube.org/);
- a specially modified version of the SonarQube C++ Community Plugin - [https://github.com/SonarOpenCommunity/sonar-cxx](https://github.com/SonarOpenCommunity/sonar-cxx);
- the SAFacilitator – an application largely simplifying the static analyzers usage and the integration of their results into SonarQube – more info @ [https://www.spazioit.com/pages_en/sol_inf_en/code_quality_en/safe-toolset/](https://www.spazioit.com/pages_en/sol_inf_en/code_quality_en/safe-toolset/)
The development of the SAFe Toolset has been funded by the European Space Agency Contract # RFP/3-15558/18/NL/FE/as.
Future/Current Activities

June 2019
Future/Current Activities

- Spazio IT has just started working on Software Verification and Validation and Artificial Intelligence (especially Machine Learning). This research work is active on two complementary fronts:
  1. how to verify and validate AI software
  2. how to improve the “traditional” verification and validation activities with the adoption of AI techniques.
- Some new generations of static analyzers may be based on AI techniques.

June 2019
Thank you for your time!