



***Δ «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>

Agenda



June 2019

2

Agenda



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

Need for Speed



June 2019

4

Need for Speed



- The size of software codebases is increasing dramatically:

Year	System	Size
1974	F16A Plane	135 K
1981	Space Shuttle PFS	400 K
2008	ESA ATV	1 M
2012	NASA Curiosity	2.5 M
2012	F35 Plane	10 M
Nowadays	Car	10-150 M

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

Need for Speed



The screenshot displays the SonarQube web interface. The browser address bar shows the URL `sonarsrv.spazioit.com/projects?sort=-analysis_date`. The interface includes a navigation menu with 'Projects', 'Issues', 'Rules', 'Quality Profiles', and 'Quality Gates'. A search bar is present for projects and files. The main content area shows a list of five projects, each with a 'Passed' status and various quality metrics:

Project Name	Status	Bugs	Vulnerabilities	Code Smells	Coverage	Duplications	Last Analysis	Size
Node.js	Passed	6.3k (D)	49 (C)	268k (A)	0.0%	3.6%	May 31, 2019, 4:51 PM	1M (XL)
Naviserver	Passed	661 (D)	0 (A)	25k (A)	0.0%	0.2%	May 23, 2019, 1:32 PM	50k (M)
Crazyfly_EXPLODED	Passed	99 (C)	0 (A)	24k (A)	0.0%	28.0%	May 3, 2019, 7:16 PM	511k (XL)
Crazyfly	Passed	521 (D)	2 (C)	45k (B)	0.0%	3.0%	May 3, 2019, 5:59 PM	62k (M)
CCSDS_File_Delivery_Protocol	Passed	0 (A)	0 (A)	5.8k (B)	0.0%	1.4%	March 30, 2019, 6:38 PM	9.7k (S)

The left sidebar contains filters for Quality Gate (Passed: 5, Warning: 0, Failed: 0), Reliability (A: 1, B: 0, C: 1, D: 3, E: 0), Security (A: 3, B: 0, C: 2, D: 0, E: 0), Maintainability (A: 3, B: 2, C: 0, D: 0, E: 0), and Coverage (≥ 80%: 0, 70-80%: 0, 50-70%: 0, 30-50%: 0, < 30%: 5, No data: 0).

Need for Speed



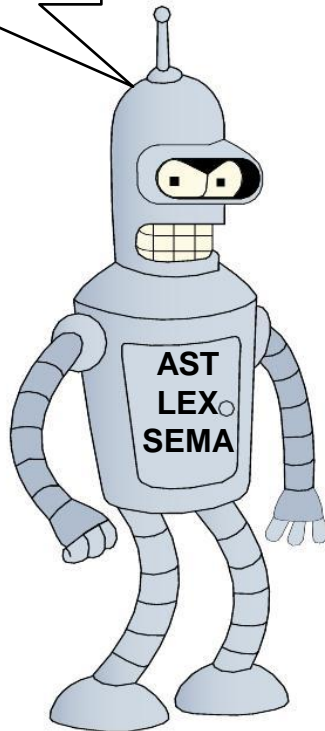
- Deep vs. Shallow Parsing

- Unforgiving vs. Forgiving Parsing

Libraries, Libraries and Again Libraries

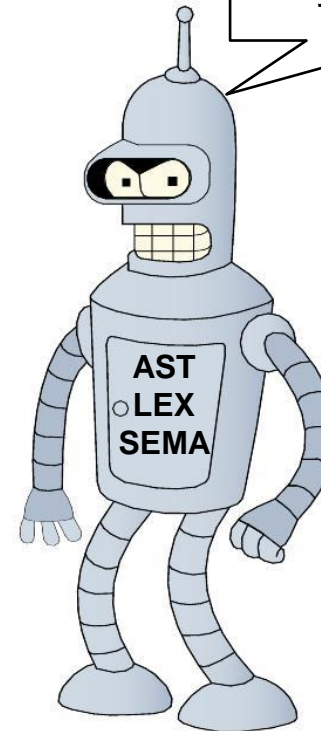


Are we siblings?



Static Analyzer

I don't know!
Do you use my libraries?



Compiler

Libraries, Libraries and Again 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?

Libraries, Libraries and Again Libraries



- PC-Lint does not support the latest C/C++ Standards.
- Frama-C Semantic Analyzer cannot process all C/C++ constructs.
- Cppcheck sometimes stops when “digesting” “strange” codebases (e.g. Brotli).
- Ada ASIS does not support Ada 2012 (but the GNAT compiler does).
- In the Ada “libadalang” GitHub 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...

Libraries, Libraries and Again Libraries



- “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.

Libraries, Libraries and Again 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:\naviserver-src\clang-visitor.cc - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
clang-visitor.cc
29
30 CXCursor cursor = clang_getTranslationUnitCursor(unit);
31 clang_visitChildren(
32     cursor,
33     [](CXCursor c, CXCursor parent, CXClientData client_data)
34     {
35         if (clang_Location::FromMainFile (clang_getCursorLocation (c))) {
36             if (strcmp ("Function", clang_getCursorKindSpelling (clang_getCursorKind (c))) == 0) {
37                 printf ("Function found in file %s\n", clang_getCString (clang_getCursorLocation (c)));
38             }
39         }
40         return CXChildVisit_Continue;
41     },
42     nullptr);
43
44 clang_disposeTranslationUnit (unit);
45
```

libclang

C++ source file length : 1,283 lines : 47 Ln : 30 Col : 13 Sel : 0 | 0 Windows (CR LF) UTF-8 IN

Libraries, Libraries and Again Libraries



- “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



NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

+ABOUT NASA | +LATEST NEWS | +MULTIMEDIA | +MISSIONS | +WORK FOR NASA

+ NASA Home
+ Ames Home
+ Intelligent Systems Division

IKOS

- + Home
- + Research Themes
- + Projects
- + Publications
- + RSE
- + Internal Systems Help
- + Missions
- + Research
 - + AdaStress
 - + AdvoCATE
 - + CoCoSim
 - + fret
 - + IKOS
 - + MARGInS
 - + MESA

Robust Software Engineering

IKOS: Inference Kernel for Open Static Analyzers

The objective of this project is to perform scalable, precise static analysis of C and C++ code for aviation.

To this end, we have developed a tool, IKOS, that relies on the theory of Abstract Interpretation for analyzing C and C++ code. IKOS is really a framework for static analysis based on abstract interpretation. It relies on the LLVM framework for its front-end and implements various analyses based on its own library of abstract interpretation components (forward iterators, abstract domains, ...).

Latest News

October 2018: IKOS 2.0 released on GitHub.

Other information

IKOS website
IKOS downloads

Active Members

Maxime Arthaud
Guillaume Brat
Nija Shi

Past Members

Arnaud Hamon
Jorge Navas
Elodie-Jane Simms
Sarah Thompson
Arnaud Venet

libadalang



build passing

Libadalang

Libadalang is a library for parsing and semantic analysis of Ada code. It is meant as a building block for integration into other tools. (IDE, static analyzers, etc.)

Libadalang provides mainly the following services to users:

- Complete syntactic analysis with error recovery, producing a precise syntax tree when the source is correct, and a best effort tree when the source is incorrect.
- Semantic queries on top of the syntactic tree, such as, but not limited to:
 - Resolution of references (what a reference corresponds to)
 - Resolution of types (what is the type of an expression)
 - General cross references queries (find all references to this entity)

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.

If you have problems building or using Libadalang, or want to suggest enhancements, please open a [GitHub issue](#). We also gladly accept [pull requests](#)!

libadalang



libadalang/ada/language

https://github.com/AdaCore/libadalang/tree/master/ada/language

Why GitHub? Enterprise Explore Marketplace Pricing Search Sign in

AdaCore / libadalang

Watch 21 Star 47 Fork

Code Issues 12 Pull requests 2 Projects 1 Insights

Branch: master libadalang / ada / language / Create new file Find file

Roldak and raph-amiard S419-013: Make referenced_decl return the instantiation. Latest commit 39c97c3 6

..

<code>__init__.py</code>	Remove all unicode_literals imports from __future__	2 y
<code>ast.py</code>	S419-013: Make referenced_decl return the instantiation.	3 c
<code>documentation.py</code>	Reject aggregate projects in the project unit provider	3 mo
<code>grammar.py</code>	Sort imported entities in all Python scripts	7 c
<code>lexer.py</code>	lexer: replace uses of the "ada_lexer.patterns" helper	3 mo

<https://github.com/AdaCore/libadalang/commit/a748e6e6a4d332e6fcf1866c2485b2069e9f2c>



Libadalang and ASIS

ASIS is widely used for static analysis of Ada code, and is an ISO standard. It is still the go-to tool if you want to create a tool that analyses Ada code. Also, as explained above, Libadalang is not mature yet, and cannot replace ASIS in tools that require semantic analysis.

However, there are a few reasons you might eventually choose to use Libadalang instead of ASIS:

1. The ASIS standard has not yet been updated to the 2012 version of Ada. More generally, the advantages derived from ASIS being a standard also means that it will evolve very slowly.
2. Syntax only tools will derive a lot of advantages on being based on Libadalang:
 - Libadalang will be completely tolerant to semantic errors. For example, a pretty-printer based on Libadalang will work whether your code is semantically correct or not, as long as it is syntactically correct.
 - Provided you only need syntax, Libadalang will be much faster than ASIS' main implementation (AdaCore's ASIS), because ASIS always does complete analysis of the input Ada code.
3. The design of Libadalang's semantic analysis is lazy. It will only process semantic information on-demand, for specific portions of the code. It means that you can get up-to-date information for a correct portion of the code even if the file contains semantic errors.
4. Libadalang has bindings to C and Python, and its design makes it easy to bind to new languages.
5. Libadalang is suitable to write tools that work on code that is evolving dynamically. It can process code and changes to code incrementally. Thus, it is suitable as an engine for an IDE, unlike AdaCore's ASIS implementation.



- Interesting related projects:
 - **libadalang-tools** - Libadalang-based tools
 - **lal-checkers** - Libadalang-based code checking infrastructure
 - **ada_language_server** - prototype implementation of the Microsoft Language Server Protocol for Ada/SPARK
 - **langkit** - Language creation framework.

Clang / LLVM – SonarQube Integration



The image displays a code editor window on the left and a SonarQube web interface on the right. The code editor shows the following C code:

```
93 static Basic b;  
94 static MeasData m;  
95  
96 // Only use on 0-terminated strings!  
97 static int skip_to_next(char ** sp, <  
98 int steps;  
99  
100 while (ch != 0 && (**sp) != ch) {  
101     (*sp)++;  
102     steps++;  
103 }  
104
```

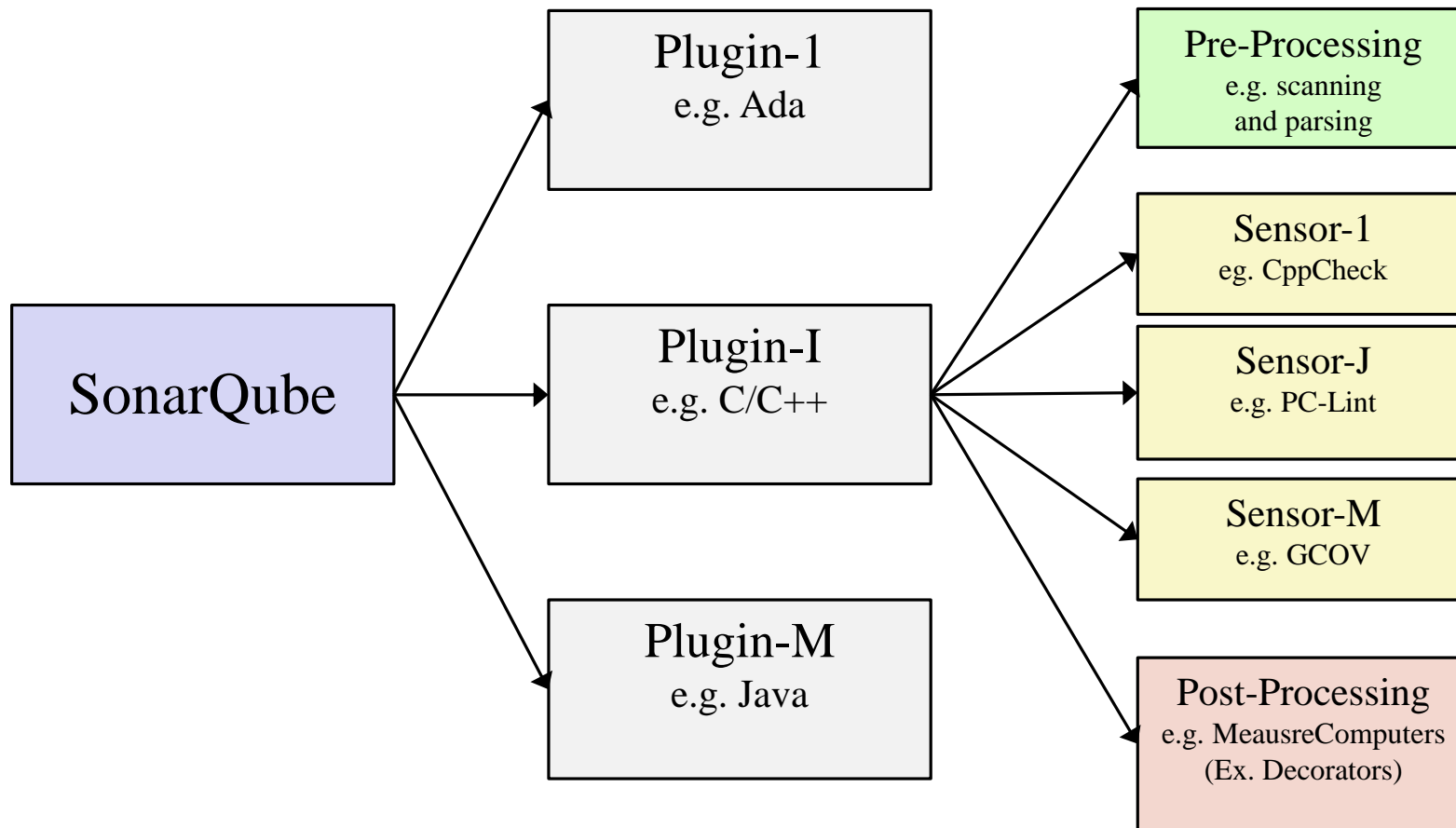
Yellow callouts in the code editor highlight specific lines:

- 3 ← 'steps' declared without an initial value →
- 4 ← Assuming the condition
- 5 ← Loop condition is true. Entering loop body
- 6 ← The expression is an uninitialized value. T

The SonarQube interface shows a bug report for the file `gtgps.c`. The bug title is "The expression is an uninitialized value. The computed value will also be garbage". The severity is "Major" (+6). The bug is located at line 101 of the file. The SonarQube interface also shows a list of steps for the bug:

- 1 Calling 'skip_to_next'
- 2 Entered call from 'gpgsaParser'
- 3 'steps' declared without an initial value
- 4 Assuming the condition is true
- 5 Entering loop body
- 6 The expression is an uninitialized value. The computed value will also be garbage

SonarQube / Plugins / Sensors



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

Clang / LLVM – SonarQube Integration



The image shows a code editor window on the left and a SonarQube web interface on the right. The code editor displays the following C code:

```
93 static Basic b;  
94 static MeasData m;  
95  
96 // Only use on 0-terminated strings!  
97 static int skip_to_next(char ** sp,  
98 int steps;  
99  
100 while (ch != 0 && (**sp) != ch) {  
101     (*sp)++;  
102     steps++;  
103 }  
104
```

Yellow callouts in the code editor highlight the following steps in the analysis:

- 3 ← 'steps' declared without an initial value →
- 4 ← Assuming the condition
- 5 ← Loop condition is true. Entering loop body
- 6 ← The expression is an uninitialized value. T

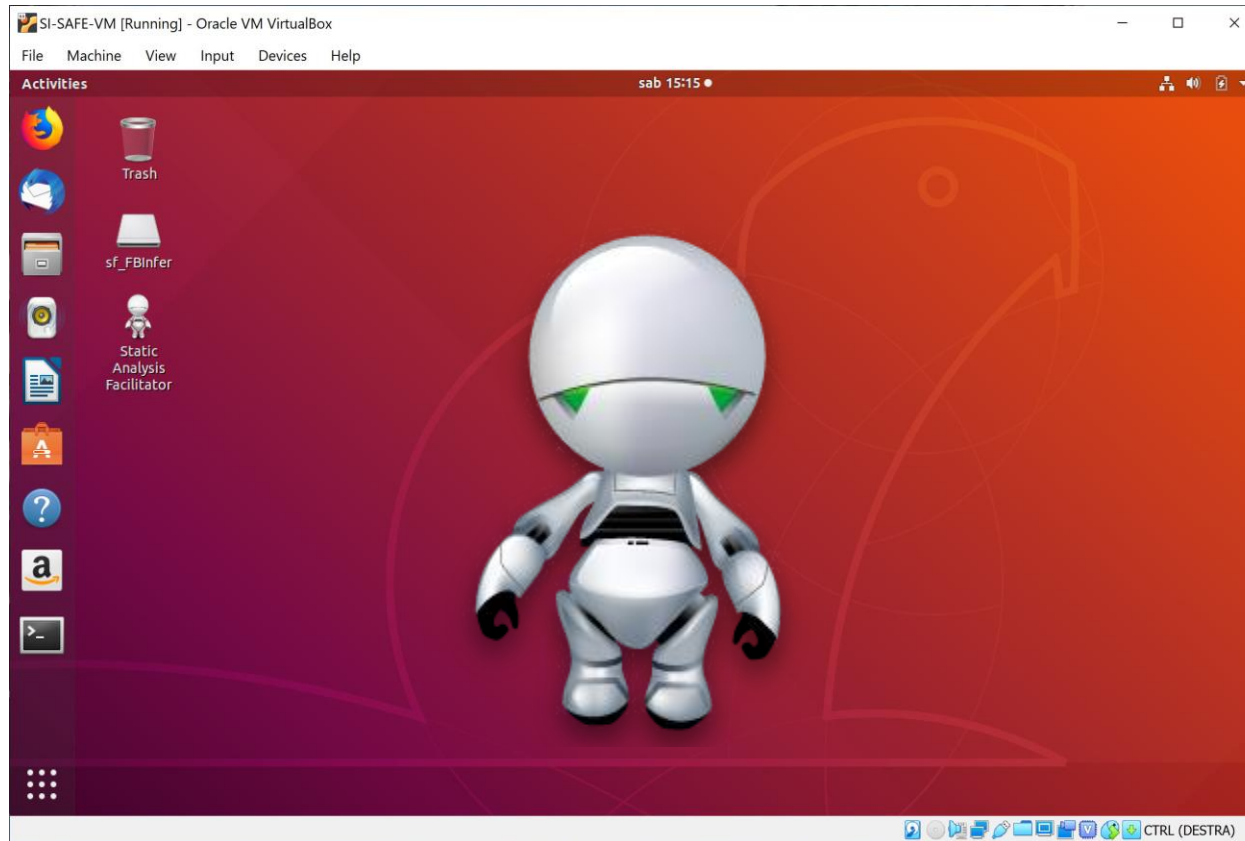
The SonarQube interface shows a bug report for the file `gtgps.c`. The bug title is "The expression is an uninitialized value. The computed value will also be garbage". The bug is categorized as "Major" and "Not assigned". The SonarQube interface also shows a list of steps in the analysis:

- 1 Calling 'skip_to_next'
- 2 Entered call from 'gpgsaParser'
- 3 'steps' declared without an initial value
- 4 Assuming the condition is true
- 5 Entering loop body
- 6 The expression is an uninitialized value. The computed value will also be garbage

The SonarQube interface also shows a list of steps in the analysis:

- 2 static int skip_to_next(char ** sp, const char ch) {
- 3 int steps;
- 4 5 while (ch != 0 && (**sp) != ch) {
- 6 steps++;

SAFe Toolset

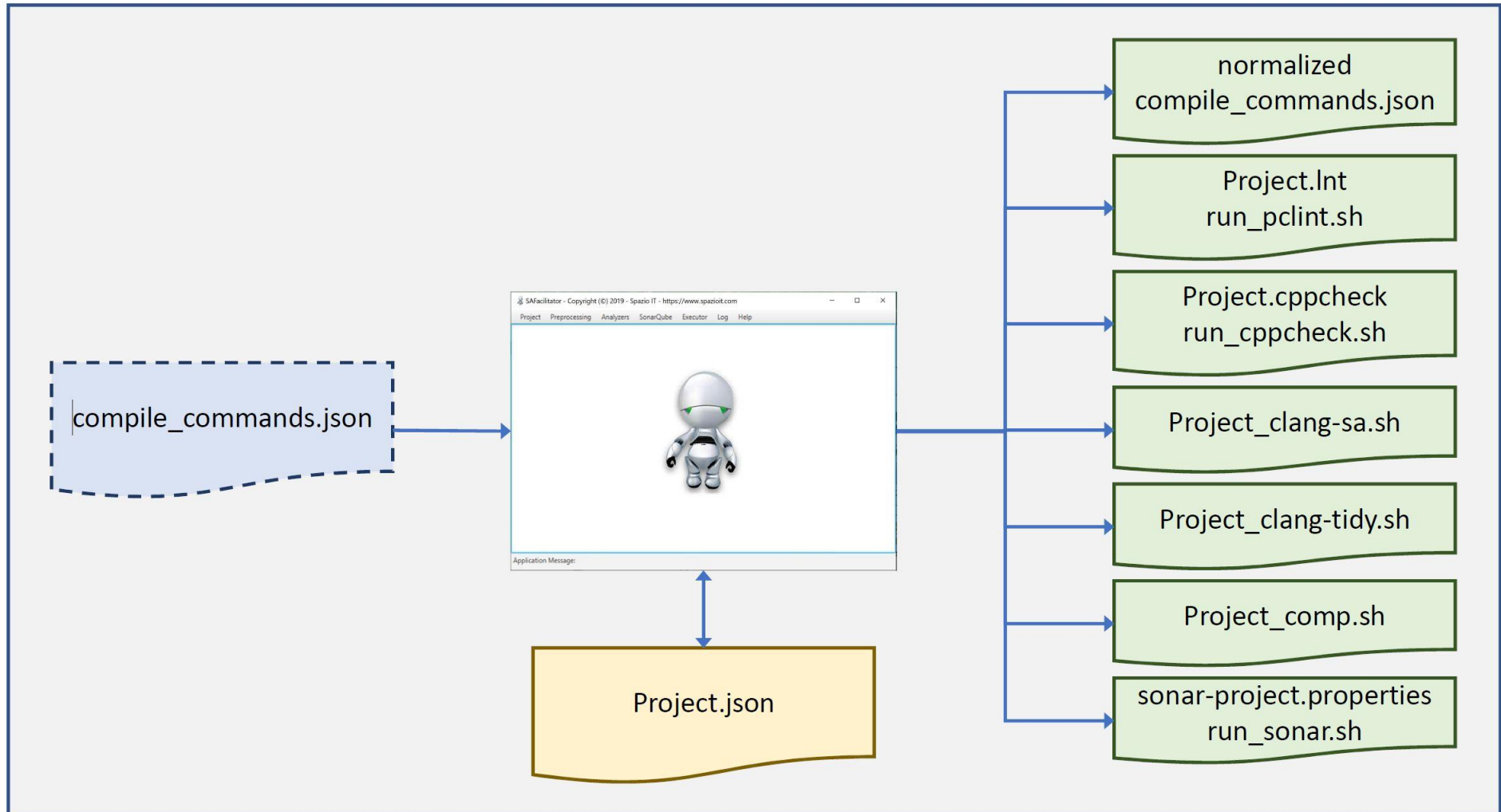




The screenshot shows a web browser window with the following content:

- Browser tab: JSON Compilation Data
- Address bar: <https://clang.llvm.org/docs/JSONCompilationDatabase.html>
- Page title: Clang 9 documentation
- Section title: JSON COMPILATION DATABASE FORMAT SPECIFICATION
- Breadcrumbs: « [How To Setup Clang Tooling For LLVM](#) :: [Contents](#) :: [Clang's refactoring engine](#)
- Section title: **JSON Compilation Database Format Specification**
- Text: This document describes a format for specifying how to replay single compilations independently of the build system.
- Section title: **Background**
- Text: 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:
- List-Group:
 - 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 end-user tools 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





- 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:
 - **cppcheck** – v. 1.87 - <http://cppcheck.sourceforge.net/> - a C/C++ static analyzer.
 - **Clang** – v. 9.0.0 - <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/> - 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/> - 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/> - Native
 - **Clang** Version 9.0.0 - - <https://clang.llvm.org> - Native and Cross (Multiplatforms - use the command “llc --version” to see the supported architectures).
 - **BCC2**: Bare-C Cross-Compiler System for LEON2/3/4 GCC 7.2.0 - <https://www.gaisler.com/> - Cross.
 - **GNU Arm Embedded Toolchain** - v. 5-2016-q3 - <https://launchpad.net/gcc-arm-embedded> - Cross.

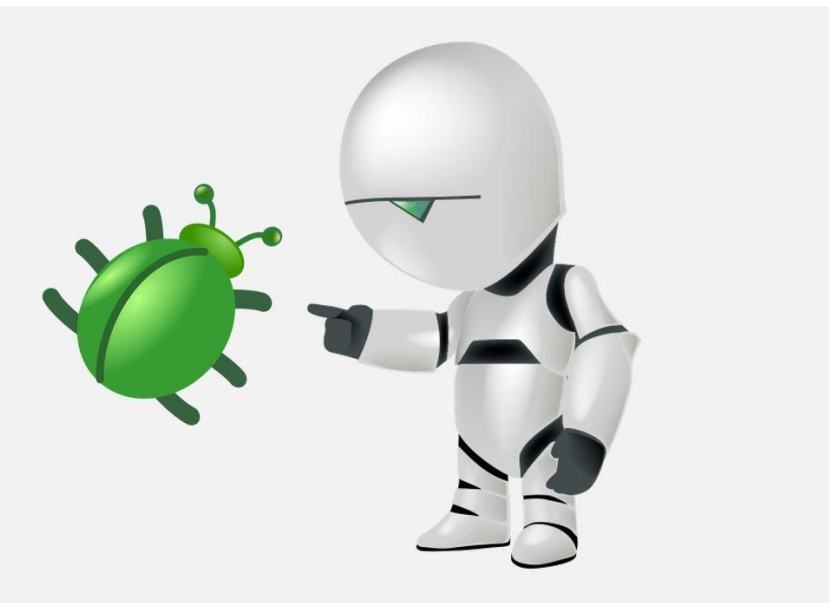
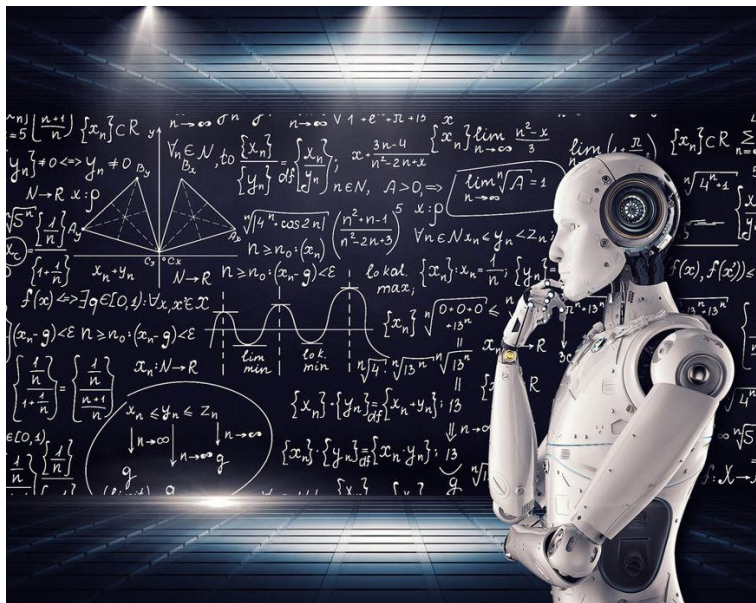


- Should a user need to work on a codebase not supported by the provided build environments, she would need to install the corresponding compilation toolchain.
- Additionally Spazio IT has complemented the SAFe Toolset with:
 - a specially **modified version of SonarQube** - <https://www.sonarqube.org/> ;
 - a specially **modified version of the SonarQube C++ Community Plugin** - <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/



- 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



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.

Thank you for your time!

