2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

Integrating Static Code Analysis Toolchains

Matthias Kern*, Ferhat Erata§¶, Markus Iser‡, Carsten Sinz†, Frederic Loiret‡, Stefan Otten*, and Eric Sax*,
*FZI Research Center for Information Technology, Karlsruhe, Germany
†Karlsruhe Institute of Technology, Institut für Theoretische Informatik, Karlsruhe, Germany
‡KTH Royal Institute of Technology, Embedded Control Systems, Stockholm, Sweden
§Yale University, Department of Computer Science, New Haven, USA
¶UNIT Information Technologies, Research & Development, Izmir, Turkey
*{*mkern, otten, sax*}*@fzi.de* †{*markus.iser, carsten.sinz*}*@kit.edu* ‡*floiret@kth.se* §*ferhat.erata@yale.edu*

***Abstract*—This paper proposes an approach for a tool-agnostic and heterogeneous static code analysis toolchain in combination with an exchange format. This approach enhances both traceability and comparability of analysis results. State of the art toolchains support features for either test execution and build automation or traceability between tests, requirements and design information. Our approach combines all those features and extends traceability to the source code level, incorporating static code analysis. As part of our approach we introduce the "ASSUME Static Code Analysis tool exchange format" that facilitates the comparability of different static code analysis results. We demonstrate how this approach enhances the usability and efficiency of static code analysis in a development process. On the one hand, our approach enables the exchange of results and evaluations between static code analysis tools. On the other hand, it enables a complete traceability between requirements, designs, implementation, and the results of static code analysis. Within our approach we also propose an OSLC specification for static code analysis tools and an OSLC communication framework.**

***Index Terms*—Traceability, Interoperability, Static Analysis, OSLC**

I. Introduction

Highly automated vehicles with more than hundred electrical control units (ECUs) and millions of lines of code are good examples of safety-critical, complex systems [2], [3]. In the future, with the technological advances in autonomous driving, the complexity of those highly automated mobility systems will increase further in the number of sensors, communication pathways, and functionality.

Yet, there are many tools without any linkage to other ones building so-called "islands of information" [4]. This means the data produced by such tools has no traceable connection between each other. In order to support the development of highly automated mobility systems in a safe and secure manner, toolchains that give the possibility to trace and exchange all design artifacts over the complete life-cycle are needed. Such toolchains would enable a direct exchange of information between different tools and thus enhance the traceability between the different sources of information. Standardized exchange formats are crucial to the creation of tool adapters which increase the interoperability among them and enhance the comparability of their outputs. Toolchains that are not constrained to a specific set of tools and that support the replacement of them, are so-called tool-agnostic toolchains.

This work was funded by German Federal Ministry of Education and Research (BMBF) under grant #01IS15031A as part of ASSUME [1] project.

Today, it is difficult to reuse configurations and to compare reports of different static code analysis tools (SCAT) since they mostly use a proprietary data format for both the analysis results and the analysis configuration. Furthermore, they have its own strengths and developers must often aggregate the analysis results of different analyzers to form an overall picture of program quality [5]. However, without standardized exchange formats, it is not easy to combine their strengths. Additionally, it is very common for tool vendors to offer linkage between their own products, especially for web-based ones; nevertheless, tools of different vendors are required within a toolchain. Besides, there are many tools without a possibility to link their data easily between each other. To overcome aforementioned limitations, we propose a tool-agnostic and heterogeneous toolchain for SCAT.

The rest of the paper begins by presenting the background and related-work of our approach in Section II. In Section III we give an overview of concepts through an exemplary use-case for "ASSUME Static Code Analysis Tool Exchange Format (ASEF)", developed within the European ASSUME ITEA3 Project [1]. In Section IV we introduce technical concepts implementing the toolchain in which an adapter communication framework based on OSLC and the ASEF Format are presented, as well as an approach for static code analysis automation that supports traceabilty to design artifacts. Finally we give a conclusion and future work in Section V.

II. Background and Related Work

In this section, we present the basic concepts and technologies that are necessary to understand our work. In Section II-A, we give a quick introduction into static code analysis with three small examples to motivate the necessity of static code analysis. In Section II-B, we describe the current literature on the traceability research to position our work in this area. In section II-C, we present OSLC, which builds technologically the base of our approach and is used to set up our toolchain that addresses system development. Since one of the main contributions of this work is the so-called ASSUME SCA tool exchange format (ASEF), in Section II-D, we explain a closely related standard, "Static Analysis Results Interchange Format" (SARIF) and discuss how our format is complementary to the SARIF. Finally, in Section II-E, we present common continuous integration (CI) technique that today does not support traceability to design artifacts and comparability of different static code analysis tools.

A. Static Analysis

Static code analysis involves methods and algorithms to automatically proof the absence of specific types of unwanted behavior in a piece of code. Static analysis tools can calculate a combination of input parameters and an execution trace that lead to an invalid state in a program. Such invalid states might include *undefined behavior*, the violation of *custom assertions* or the access of *uninitialized memory*.

1) Undefined Behavior: Unspecified semantics where the behavior of a programming language becomes unpredictable are commonly known as *undefined behavior*. Such states are unwanted and should not be reachable at all. In the piece of code shown in example 1, the removal of lines 2 and 3 leads to the reachability of an undefined state (considering the semantics of the C programming language).

Example 1: Undefined Behavior

```
1 int $z \leftarrow a - b$
2 if $z =$ MIN_INT then
3     handle_invalid_state()
4 int $y \leftarrow -z$
```

2) Custom Assertions: Code optimization can lead to obfuscated pieces of code that are hard to comprehend and verify. There are scenarios, where code optimization is indispensable. Example 2 shows how developers can use a *custom assertion* to use static analysis to show that an optimized piece of code still behaves exactly the same as the unoptimized variant.

Example 2: Function Equivalence

```
1 int $a \leftarrow$ foo()
2 int $b \leftarrow$ foo_optimized()
3 static_assert($a = b$)
```

3) Uninitialized Memory: Working with old or unstructured low-level code can be a challenge with respect to memory management. Functional extensions might lead to non-trivial bugs which can not easily be discovered. Example 3 is an abstract representation of a common situation.

Example 3: Access Uninitialized Memory

```
1 if init-condition then
2     initialize memory
3 do some processing and access memory
```

While `init-condition` (see line 1) might hold whenever needed in the original version of the software this property might get lost during a sequence of extensions and patches. Static code analysis tools can automatically trace execution paths to states where uninitialized memory is accessed.

B. Traceability

Regarding the industrial tools and technologies on traceability, modeling tools such as EMF [6] and SysML [7], requirement interchange standard (ReqIF [8]) and management tools such as RMF[1] and IBM Rational DOORS [9] provide some automated or manual means to specify and manage traceability. However, none of them provides integration with static analysis code analysis tools, especially on a heterogeneous development and design environment.

C. Open services for Lifecycle Collaboration (OSLC)

Open services for Lifecycle Collaboration (OSLC)[2] is an open community that defines specifications to link the data of different tools, used in the Application Lifecycle Management (ALM) [10] and Product Lifecycle Management (PLM) [11], in order to directly support traceability. The OSLC specifications build on REST [12], the W3C Resource Description Framework (RDF) and Linked Data[3].

OSLC offers specifications for the requirement-management, the quality-management (QM) and the architecture-management. With the architecture-management specification, data from modeling-tools can be mapped to resources. Data from testing tools can be mapped with the help of the QM specification [13]. However, a specification for mapping results of static code analysis tools is missing. Therefore, a resource definition based on the QM specification and the ASEF format have been created as part of our approach. This specification is shown more in detail in the Section IV-C.

For several tools, there are already OSLC adapters available, like the Matlab Simulink integration from Axel Reichwein[4] or for Bugzilla, a bug-tracking tool[5].

D. Static Analyis Results Interchange Format (SARIF)

The Static Analysis Results Interchange Format (SARIF) is a standardized interchange format that enables the aggregation of results of different analysis tools. SARIF was originally developed by Microsoft and is currently being standardized by OASIS in the OASIS SARIF Technical Committee. The format addresses a variety of analysis tools that can indicate problems related to program qualities such as correctness, security, performance, compliance with contractual or legal requirements, compliance with stylistic standards, understandability, and maintainability. There are several tools available for the programming language C# as SDKs or Converters. Furthermore, there is a Viewer for Visual Studio extension [6]. SARIF enhances the usability by combining and comparing the result in a more easier way than the several competing static analysis tools [7].

[1]https://www.eclipse.org/rmf/
[2]http://open-services.net/software/
[3]http://open-services.net/
[4]https://github.com/ld4mbse/oslc-adapter-simulink
[5]http://wiki.eclipse.org/Lyo/BuildOSLC4JBugzilla
[6]https://sarifweb.azurewebsites.net/
[7]https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif

E. Continuous integration with static code analysis tools

Current static code analysis tools like Astrée[8] from AbsInt or Coverty Scan from Synopsys offer a Jenkins[9] plugin that allows using these tools before the build process starts. Jenkins itself enables a continuous integration. The integration process, including testing and a build process, is normally triggered by a REST request. This request is normally sent out from a Git-repository manager, like GitHub or GitLab, after code was pushed to a specific git repository. However, traceability from testing results to requirements or other design artefacts is not supported through Jenkins. Furthermore, Jenkins plugins are specific for each tool, so there is no standard that enables a plugin for different static code analysis tools and offers comparability between different analysis tools.

III. Concept and Use-Case

To motivate our approach we present an illustrative example for the development of a functionality for a direction indicator lamp. This functionality could be run on an electronic control unit (ECU). Direction indicator lamps or informally "blinkers" or "flashers" are blinking lamps mounted near the left and right front of a car and can be activated by the driver at a time to advertise intent to turn or change lanes towards that side. For direction indicator lamps exist regulations like the *E/ECE/324/Rev.1/Add.47/Rev.12 - E/ECE/TRANS/505/Rev.1/Add.47/Rev.12* [10]. To find all necessary regulations and requirements, requirement engineering tools as shown in Figure 2 can be used. In this regulation there is a requirement in section 6.5 that says *"The light shall be a flashing light flashing 90±30 times per minute"*. To fulfill this requirement the functionality of the direction indicator lamp is designed with a *stateflow chart* (cf. Figure 1). The design of the functionality could be done with design tools as shown in Figure 2.

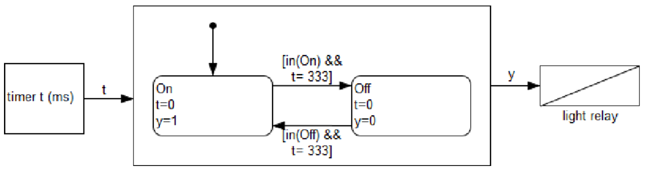

Fig. 1. Direction indicator lamp for a car.

After the system design the code can be written or generated during the implementation activity. A piece of software code for the functionality of the direction indicator lamp could look like that shown in Listing 1.

Here, we assume to have a hardware timer that is increased every millisecond by one, and its value can be read out using function `getTimer()`, which returns a signed short. To achieve a blinking frequency of 1.5 Hz, the light should be switched every 333 ms. To make sure that the implementation satisfies the requirement, we have added assertions (via `static_assert`) to make sure that a switch occurs after between 250 ms and 500 ms and that the time increases in each iteration of the loop.

```c
typedef enum { Off, On } state_t;
typedef short time_t;

extern time_t getTimer();
extern void setIndicatorLamp(state_t s);

int timerExpired(time_t start, time_t end, time_t diff)
{
    return end-start > diff;
}

void process()
{
    time_t startTime = getTimer(), currentTime;
    state_t light = Off;
    while (1) {
        currentTime = getTimer();
        static_assert(currentTime - startTime >= 0);
        if (!timerExpired(startTime, currentTime, 333)){
            continue;
        }
        if (light == Off) {
            setIndicatorLamp(light = On);
        } else {
            setIndicatorLamp(light = Off);
        }
        static_assert(currentTime - startTime >= 250);
        static_assert(currentTime - startTime <= 500);
        startTime = currentTime;
    }
}
```

Listing 1. Direction indicator lamp C Code.

Due to an integer overflow bug, the implementation will not work in all cases, in particular on a 32-bit platform. E.g., if `startTime = 32700` and `currentTime` has a negative value due to an overflow, then end - start in function `timerExpired()` will be a large negative value (due to integer promotion no overflow occurs in the subtraction), and it will take a long time (approx. 65 seconds) until a timer expiration is reported.

This kind of overflow bug is not only of academic interest, but also of practical importance, as an incident from 2015 shows: Engines of the Boeing 787 Dreamliner could fail due to loss of electric power after 248 days of continues operation[11]. The fault was caused by a timer-related integer overflow bug similar to that of the example above.

The analysis of the code from Listing 1 belongs to analysis activities, and many static analysis tools will be able to find the bug in the implementation. Output from an analysis tool might look as in Example 4.

All these artifacts of our example should be traceable within the proposed toolchain. Therefore we give here an exemplary overview of possible tools which allows us to manage and produce the necessary results. An overview of this toolchain is given in Figure 2. To make a proof of concept we implemented

[8]https://www.absint.com/astree/index.htm
[9]https://jenkins.io/
[10]https://www.unece.org/fileadmin/DAM/trans/main/wp29/wp29regs/2015/R048r12e.pdf
[11]See, e.g., https://arstechnica.com/information-technology/2015/05/boeing-787-dreamliners-contain-a-potentially-catastrophic-software-bug.

Example 4: Analysis results.

```
Assertion in line 18 failed:
startTime = 32452
currentTime = -32684
```

a part of this concept (red shaded box in Figure 2). As a technology to implement the traceability links among artifacts we employ Open Services for Lifecycle Collaboration (OSLC) (cf. Section II). The requirement analysis activities can be

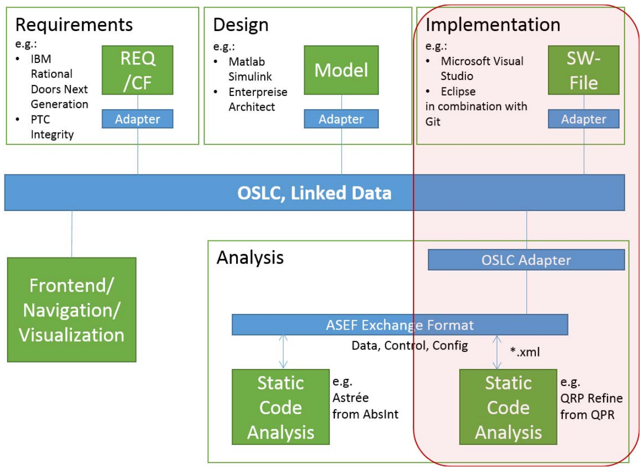

Fig. 2. Toolchain

supported with IBM Rational Doors Next[12] or PTC Integrity[13]. The design activity could be supported with Matlab Simulink from MathWorks[14] or Enterprise Architect from SparxSystems[15]. With Matlab Simulink, the behaviour of systems can be modeled and simulated. The architectural description of a system can be described with Enterprise Architect, which supports Unified Modeling Language (UML) and Systems Modeling Language (SysML). For the software implementation we suppose an integrated development environment like Visual Studio from Microsoft or Eclipse from the Eclipse Foundation. For the static code analysis, Astrée from AbsInt or QPR Refine from QPR Technologies could be used. Both tools use abstract interpretations of C code to detect runtime errors, data races or assertion violations and includes a MISRA C checker[16]. In the box "Analysis" in Figure 2 two static code analysis tools can use the ASEF format (cf. Section IV-B) to produce comparable results. However, only one analysis tool is adapted here in our toolchain.

IV. Technical Concepts

To show the process behind the scenes of our toolchain the most important technical concepts are presented in this section.

[12]https://www.ibm.com/us-en/marketplace/rational-doors
[13]https://www.ptc.com/en/products/plm/plm-products/
[14]https://www.mathworks.com/products/simulink.html
[15]https://www.sparxsystems.eu/start/home/
[16]https://www.misra.org.uk/Activities/MISRAC/tabid/160/Default.aspx

These concepts include the framework of our toolchain, the ASEF exchange format and a specification for static code analysis tools based on the OSLC quality management specification.

A. The OSLC adapter communication framework

In the following section, the framework and its communication is described. Here, only the communication between an integrated development environment (IDE) and a static code analysis tool is regarded (see red box in Figure 2). In Figure 3, the communication between the IDE and the static code analysis tool is depicted. The static code analysis tool uses a client that manages the communication. For each communication participant exists a git repository. Within these git repositories, the data is stored and version managed in a standardized format. In the case of C code the standardized format is the C code itself. In the case of the static code analysis, the analysis report is stored in the tool independent ASEF exchange format. Through the git repository, only checked in versions are linked within the tools in the toolchain. The adapters "Code Adapter P1" and "Analysis Adapter P2" parses the data from the git repositories into the Linked Data format namely "Resource Description Framework" (RDF). The adapters are triggered every time a new version is pushed into the corresponding git repositories. During parsing, they link the corresponding information of the different adapters. In this case, the information of the analysis report is linked with the C code. The participants can retrieve the linked information via the both adapters. The adapters were implemented with the help of the model based development tool "Lyo Modeller".

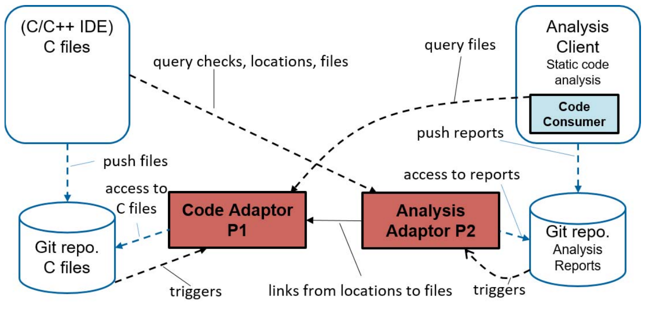

Fig. 3. Communication framework

B. The ASEF Format

The ASSUME SCA tool exchange format (ASEF) offers an XML schema for a tool-agnostic configuration format in static code analysis and for the reporting of analysis results. We developed ASEF Format aiming at tool-interoperability and facilitating well-defined check-semantics. The format is extensible and allows tool-dependent configuration. The schemes and the documentation are available on the following pages:

http://assume-project.github.io/download.html

1) ASEF Configuration Format: The ASEF configuration is split into a global part and a local part, in which the global part is intended to contain the main configuration that can be shared across multiple hosts, whereas specific hosts are able to adapt the local part of the configuration to their needs. Listing 2 shows a small example of an ASEF configuration.

The configuration allows the definition of *source modules*, *hardware targets*, *language targets* and *check targets*. Source modules define the files to analyze. Hardware targets define hardware specific properties such as pointer size or endianness. Language targets define details about the language standard and system to be checked.

```
<asef:Configuration xsi:schemaLocation="ASC3F.xsd">
<asef:GlobalConfiguration>
<asef:CommonConfiguration>
  <HardwareTargets>
   <HardwareTarget xsi:type="asef:HomogenousHardwareTarget"
        name="generic32" endianness="big" pointerSize="32"/>
  </HardwareTargets>
  <LanguageTargets>
    <LanguageTarget xsi:type="asef:CLanguageTarget" name="
        basicC11" standardRevision="C11" />
    </LanguageTarget>
  </LanguageTargets>
  <CheckTargets>
    <CheckTarget xsi:type="asef:CCheckTarge" name="base">
      <CorrectnessCheckCategory name="assert"/>
      <CorrectnessCheckCategory name="numeric"/>
      <CorrectnessCheckCategory name="controlflow"/>
      <CorrectnessCheckCategory name="mem"/>
    </CheckTarget>
  </CheckTargets>
  <ExecutionModelTargets>
    <ExecutionModelTarget xsi:type="
        asef:CSynchronousExecutionModelTarget" name="sync">
      <EntryPoints>
        <EntryPoint>main</EntryPoint>
      </EntryPoints>
    </ExecutionModelTarget>
  </ExecutionModelTargets>
  <SourceModules>
    <SourceModule name="main" rootUri="$Repository$">
    <SourceFiles>
      <SourceFile uri="example.c" id="1"/>
    </SourceFiles>
    </SourceModule>
  </SourceModules>
  <AnalysisTasks>
    <AnalysisTask name="analyzeMainSourceModule">
      <HardwareTarget>generic32</HardwareTarget>
      <SourceModule>main</SourceModule>
      <LanguageTarget>basicC11</LanguageTarget>
      <CheckTarget>base</CheckTarget>
      <ExecutionModelTarget>sync</ExecutionModelTarget>
    </AnalysisTask>
  </AnalysisTasks>
</asef:CommonConfiguration>
</asef:GlobalConfiguration>
<asef:LocalConfiguration>
  <URISubstitutionRules>
    <URISubstitutionRule token="$Repository$" substitution="
        /local/path/to/repositories">
</URISubstitutionRules>
</asef:LocalConfiguration>
</asef:Configuration>
```

Listing 2. ASEF Example Configuration

Via check targets, several check categories define which checks should be executed. One of the key-strength of ASEF lies in the precise specification of the check semantics. As different static analysis tools offer different stages of precision in various check categories there was a need to define these levels of precision. They are used in the ASEF Reports when a flaw was discovered. More details about check semantics can be found in Section IV-B2.

The execution of checks is triggered through the definition of *analysis tasks*. Analysis tasks are a combination of the previously defined targets, they specify which checks should be executed with which hardware and language configuration etc.

2) ASEF Report Format: The analysis reports involve source locations, failure traces, and check semantics.

Locations define the row and column where a fault was detected and are assigned to the checks via identifiers. It is also possible to refer from a location to another. This is useful to describe so-called macro locations, because macros can use code from other files or locations.

A check status can be safe, unsafe, undecided, warning, and syntactic violation. The check category describes the kind of fault. Table I shows a small excerpt of the hierarchy of ASEF check categories and how the categories map to those of various static analysis tools. ASEF offers well-defined check semantics and thus enables comparability of the results of the different static analysis tools.

TABLE I
ASEF CHECK SEMANTICS VS. NATIVE CATEGORIES OF VARIOUS TOOLS (INCLUDING POLYSPACE (PS) CHECK CATEGORIES, QPR CHECK CATEGORIES AND ASTREE (AS) ALARM CATEGORIES)

ASEF Category	Native Categories
numeric.overflow	PS:OVFL, AS:"Overflow in arithmetic", AS:"Initializer range"
numeric.overflow.int	QPR:arithmetic.overflow, QPR:shift.overflow
numeric.shift	PS: SHF
numeric.shift.rhs	AS:"Wrong range of second shift argument"
numeric.shift.rhs.amount	QPR:shift.by.amount
numeric.shift.rhs.negative	QPR:shift.by.negative
mem	PS:COR
mem.ptr.deref	PS:IDP, QPR:pointer.dereference
mem.ptr.deref.misaligned	AS:"Dereference of mis-aligned pointer"
mem.ptr.deref.invalid	AS:"Dereference of null or invalid pointer"
mem.ptr.deref.field	AS:"Incorrect field dereference"

C. *Proposal of an OSLC specification for static code analysis data and results exchange*

Figure 4 shows the resource definition based on both the OSLC quality management (QM) specification and the ASEF format. This definition was used to create the OSLC adapter to link the analysis information with the files and to offer an analysis case that enables the configuration for the static code analysis via OSLC. In the following, the proposed specification is explained in detail.

The orange boxes in Figure 4 represent resources that are adapted from the OSLC QM specification. The new name of the adapted resources stands in brackets behind the original name. The grey boxes represent resources based on the ASEF format. In the following, these resources are explained more in detail.

`The Analysis Case` is linked with all files that should be analyzed and contains the configuration for the static code analysis. `The Analysis Result` bunches all so-called Checks of one ASEF analysis report. There can be different Analysis Results for different versions of the source code files. One `Analysis Result` refers to a specific `Analysis`

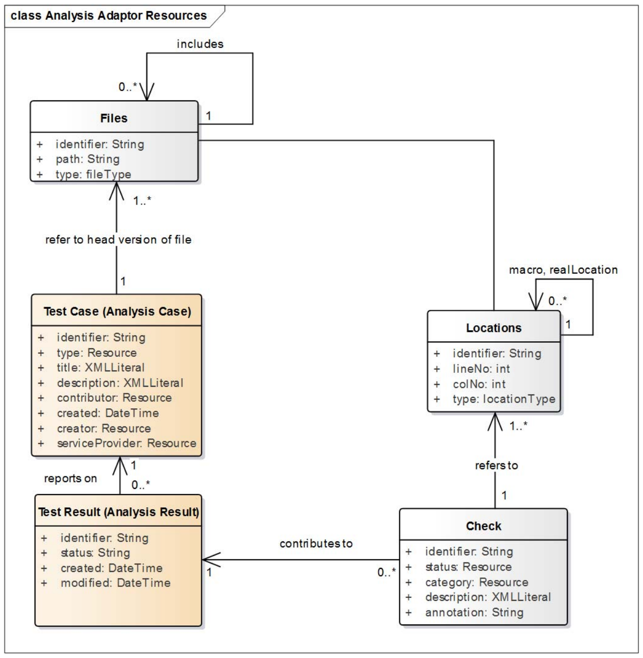

Fig. 4. Proposal static code analysis OSLC-Specification

`Case`. `Checks` contain the type of a detected fault at a specific location in the code. There is a link to the location, where the fault occurs. A `Location` gives the line and column number in the code where a problem or fault was located. The `Location` is linked with the appropriate file, or with another `Location` in the case that the `Location` is a so-called macro. Through the presented linkage the `Analysis Cases` are traceable from their results up to the source code of a specific version.

D. Approach for a static code analysis automation process

In this section, an approach for an automation process of the static code analysis within the toolchain is proposed. This process is oriented at the communication framework (cf. Figure 3) and is divided into the ten following steps: 1) A developer push files from an IDE into a git repository; 2) GitLab, that manages this git repository, triggers the analysis Client via a so-called webhook; 3) The Analysis client requests the new files from the Code Adapter P1, stores them locally and looks which analysis cases are affected through the changed files of the current commit; 4) The Analysis Client runs for each affected `Analysis Case` the analysis; 5) The Analysis Client reads the configuration from the `Analysis Cases` and changes the path for the analysis to the local stored C language files; 6) The Analysis Client starts the static analysis tool via an API; 7) The static code analysis tool analysis the files defined in the analysis configuration and produces an analysis report in the ASEF format and sends a ready acknowledgment to the analysis client; 8) The Analysis Client replaces in the analysis report the local files with the Unified Resource Identifier (URI) to the files in the Code Adapter P1 and pushes the report into the analysis repository; 9) GitLab triggers the analysis adapter, after the analysis client pushed the analysis report into the analysis repository; 10) The analysis adapter parses the information of the analysis report into Linked Data resources and links the resources of the `Locations` with the `Files` provided by the Code Adapter P1. Afterwards the developer can use a front-end to get a quick overview of the analysis results (cf. Figure 2)

V. Conclusion and Future Work

In this paper we introduced the necessity for improved traceability from the requirements downwards to static code analysis within a heterogeneous and tool-agnostic toolchain. Starting with the integration of our approach into the V-model, the state of the art and an overview about related work, we show the gaps of common solutions and motivate our approach with an use-case of a *direction indicator lamp*.

The approaches to enhance the traceability and the usability are presented as well as a technical concept with a new static code analysis exchange format, namely ASEF, a communication framework, and an OSLC specification to implement an adapter for static code analysis tools. Up to now, our implementation create linkages between code and static analysis results. For future work we are going to add new tools to prove the usability and traceability of our concepts. We demonstrated that the ASEF format brings several advantages such as the possibility to configure static code analysis tools in a uniform way. Nevertheless, we aim to show whether our approach can work also with the SARIF Standard to reach a larger community.

References

[1] ITEA, "ASSUME: Affordable Safe & Secure Mobility Evolution," https://itea3.org/project/assume.html, Sep 2015.

[2] G. Macher, M. Bachinger, and M. Stolz, "Embedded multi-core system for design of next generation powertrain control units," in *2017 13th European Dependable Computing Conference (EDCC)*, Sept 2017, pp. 66–72.

[3] J. Mössinger, "Software in automotive systems," *IEEE Software*, vol. 27, no. 2, pp. 92–94, March 2010.

[4] J. El-khoury, D. Grdr, and M. Nyberg, "A model-driven engineering approach to software tool interoperability based on linked data," vol. 9, pp. 248–259, 01 2016.

[5] A. Fatima, S. Bibi, and R. Hanif, "Comparative study on static code analysis tools for c/c++," in *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, Jan 2018, pp. 465–469.

[6] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[7] M. Soares and J. Vrancken, "Model-driven user requirements specification using SysML," *Journal of Software*, vol. 3, no. 6, pp. 57–68, 2008.

[8] A. Graf, N. Sasidharan, and Ö. Gürsoy, *Requirements, Traceability and DSLs in Eclipse with the Requirements Interchange Format (ReqIF)*. Springer Berlin Heidelberg, 2012, pp. 187–199. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25203-7_13

[9] IBM, "Rational DOORS: A requirements management tool for systems and advanced IT applications," 2011.

[10] D. Chappell, *What is Application Lifecycle Management?* Chappell & Associates, 2008.

[11] M. Eigner and R. Stelzer, *Product Lifecycle Management : ein Leitfaden für Product Development und Life Cycle Management*, 2nd ed. Berlin: Springer, 2009.

[12] S. Patni, "Pro restful apis : Design, build and integrate with rest, json, xml and jax-rs," Berkeley, CA, 2017.

[13] "Oslc specifications," http://open-services.net/specifications/, 2018, accessed: 2018-08-29.