OVERLORD: A C++ overloading inspector

Botond István Horváth[0009-0006-9997-0016], Richárd Szalay[0000-0001-5684-5158], Zoltán Porkoláb[0000-0001-6819-0224]
Department of Programming Languages and Compilers
Faculty of Informatics, Institute of Computer Science
ELTE Eötvös Loránd University
Budapest, Hungary
{*ak2u8g,szalayrichard,gsd*}*@inf.elte.hu*

***Abstract*—Function overloading is a well-known technique available in most major programming languages, including C++, to facilitate compile-time polymorphism. Due to the complex requirements imposed by the C++ Language Standard, the behaviour of overload resolution can be surprising even for experienced C++ developers. This is exacerbated by the fact that if an unexpected function is selected in an overloaded situation, most compilers do not explain why. In addition, an overabundance of candidates can be a compilation time bottleneck. Despite its widespread usage, there are not many tools to help programmers analyse overload usage in a software project.**

We developed an overloading inspector tool, OVERLORD, based on the open-source LLVM/Clang Compiler Infrastructure. With OVERLORD developers can list the possible candidate functions for a call site, and obtain step-by-step reasoning about the candidate selection process. Additionally to its comprehension functionality the tool also provides profiling data on overload resolution times to help library authors streamline the set of available overloads for improving the compilation performance.

***Index Terms*—C++ programming language, function overloading, object-oriented programming, abstract syntax tree *(AST)*, static analysis, Clang**

I. INTRODUCTION

Function overloading and *operator overloading* allow software projects to offer polymorphism – most often *ad-hoc polymorphism* – for function names and operator tokens that appear visually the same in client code. Using overloading, the call sites of a function can appear as trivial as, e.g., "`std::pow(x,y);`", and the various overloads can hide different implementations based on the type of the parameter(s). Overloading is widely supported in modern programming languages, including C++, with varying degrees of potential customisation. Operator overloading was one of the first language elements added to C++ feature set, implemented as early as 1984 [1], [2].

Just like with all language elements, the respective Standard [3] or primary design document of the language clearly defines the rules by which *overload resolution* works. Unfortunately, however, overloading is not a language element that is exercised in isolation. There is an intricate sequence of name resolution [4], template look-up [5], access control, type matching, or potential type conversions – during which additional user-defined and, potentially, themselves overloaded conversion functions may be exercised [6] – performed prior to and during overload resolution. The interplay of the rules poses a **non-trivial complexity** for developers – as shown in Section III, irrespective of their expertise – to overcome in order **to accurately understand** the workings of the language, in order to understand both why a particular overload was selected, and, conversely, if overload resolution fails, the reasons behind the failure. In addition, performing overload resolution may **consume a considerable amount of time** during compilation [7]. Our experiments – detailed in Section V – showed that, on average, 28.13% of the time spent on semantic analysis is spent because of overload resolution, which becomes 34.21% if we exclude the 4 projects that took less than 5 minutes to compile (the others took more than 30 minutes, and all had a ratio between 28.6% and 38.88%). The highest ratio was 38.88%. Allowing developers, and especially *library authors*, to better understand overload resolution times helps improving the compilation performance.

In this paper, we introduce **OVERLORD**, a C++ overloading inspector. OVERLORD is developed as an extension of *Clang*, a free and open-source C-family compiler tool chain. First, OVERLORD is an **inspector**, allowing introspection into the compiler's overloading-related state for call sites determined interesting by the user. Second, OVERLORD performs **profiling** to measure both the time required for overloading at interesting locations, and to extract statistics about overloading usage in projects.

The rest of the paper is organised as follows. In Section II, we introduce works related to the research and practice of overload analysis. Following, Section III shows some of the pitfalls and non-trivialities of overload resolution. Our tooling, OVERLORD, is described in Section IV. We detail the experiments and profiling results in Section V. The paper concludes, with the discussion of future work, in Section VI.

The tool is freely available in source code format at [8], and we also provide a *VirtualBox* image to facilitate testing the tool [9].

II. RELATED WORK

A large-scale empirical study on the use of function and operator overloading was conducted by Wang and Hou [10]. The authors instrumented the *GNU G++* compiler to identify the definitions and call sites of functions and applied their modified compiler to an early predecessor of the *Firefox* browser, then called *Mozilla Project*. In the *Mozilla Project* [11], 13 817 overloaded function *names* concentrated mostly on class templates (6 269) and constructors (5 793) that comprised

87% of all the overloads. However, when the *calls* were measured, constructors (43 076) and class templates (29 530) gave only 63% of the calls, and global (22 681) and member functions (16 815) were also significant.

Certain C++ libraries – especially those offering numerical computations – are regularly extended by introducing new types and implementing operator overloading for them. Such *adaptive software maintenance* can lead to unwanted compilation behaviour as discussed by Hück *et al.* [12]. The authors identified and classified problematic code constructions and proposed maintenance methods to avoid errors while still allowing operator overloading. They implemented a static analyser tool to help highlight problematic code fragments. The authors later improved their toolkit by implementing support for automatic code transformation [13].

An overuse of overloading may cause serious performance issues during compilation in the context of other programming languages, as well. In *Scala*, during the compilation process *typing* takes almost half of the compilation time. Most of this time is spent on the resolution of ***`implicit`*** *functions* [7]. Programmers may significantly reduce the build time by adhering to design guidelines, such as (1) avoiding file-level **`implicit`**s; (2) importing **`implicit`** functions' names only in the block they are used in; (3) avoiding the use of wildcards; (4) creating smaller namespaces. The *Scala 3* compiler addresses this performance issue with a new algorithm, which caches implicit resolution results more aggressively [14].

III. Overloading in C++

Function and operator overloading are part of the C++ language since the initial version [2], [15]. The motivation for operator overloading was to allow programmers to provide a more conventional and convenient notation for manipulating user-defined classes, like "**`Complex`**" or "**`string`**" [1]. Interestingly, operators on built-in types, including pointers, can not be overloaded. This led to the necessity to introduce *iterator* types in the *Standard Template Library (STL)* [16].

Overload resolution [3] is performed at function calls – including operator applications – if the function identified by the name belongs to an *overload set*, which is formed from separate functions bearing the same name, but differing in their number of type(s) of parameters, including, for member functions, the type of the "**`this`** object". Most functions can be overloaded, and it is common that constructors and operators are overloaded.

For each call site involving overloading, the compiler (1) builds the set of *candidate functions* (aka the *overload set*); (2) eliminates **non-viable** candidates; (3) analyses viable candidates to select the ***best viable*** one. Following the resolution process, the *best viable candidate* must be **uniquely selected**, or the compiler must emit an error that the call site is *ambiguous*. The exact rules of overload resolution are too complex to detail here, but several works discuss it [17], [18].

The *candidate set* is built from scratch for every resolution, and the results can not be cached, because the compiler can not make sure that the candidate set had not changed between the cached calculation and the query to currently resolve.

```
struct S {
  S()          { puts("Default.\n");    }
  S(const S&)  { puts("Copy.\n");       }
  S(S&&)       { puts("Move.\n");       }
  template <class T>
  S(T&&)       { puts("Templated.\n"); }
};

void test() {
  S s1;                          // "Default."
  S s2(1);                       // "Templated."
  const S cs = std::move(s1); // "Move."
  S s4(cs);                      // "Copy."
  S s5(s4);                      // "Templated."
}
```

Listing 1: Overload resolution during constructor calls can make unexpected decisions when multiple potential candidates are considered. The compiler selected a unique overload for each tests and output no diagnostics.

Unfortunately, letting the compiler pick an unintended overload may result in unexpected behaviour, bugs, or performance issues – e.g., in case a *move*-capable operation [19] falls back to *copy*, or a **`template`** instantiation has to take place. As long as overload resolution can determine the unique candidate – even if it is an unexpected one – compilers offer no diagnostics or reasoning as to why that selection was made.

We provide the following example which we believe are concise and complex enough to indicate that overload resolution can mislead developers.

The example is illustrated in Listing 1. In a constructor, the value the **`this`** object is being constructed from is usually considered read-only ("**`const`**"). Despite this, the constructions of `s4` and `s5` only differ in the **`const`**-ness of their single argument. For `s4`, to match the argument of the "copy" and "templated" candidates, a "**`const`** `S`" needs to convert to the parameter types "**`const`** `S&`" in both cases. For `s5`, an "`S`" expression needs to convert to either "**`const`** `S&`" ("copy") or "`S&`" ("templated").

Immediately, two questions arise: (1) Why is the construction of `s4` unambiguous? (2) Why did the compiler choose the potentially more involved "templated" *converting constructor* instead of the "usual" and available *copy constructor*? [20]

To answer question (1), we need only recognise that the Standard explicitly stipulates that if the candidates are equally viable, the *non-**`template`** one* takes priority. For question (2), the conversion "`S` → `S&`" is a *better conversion* than "`S` → **`const`** `S&`", as the former involves only a *reference binding*, but no *qualifier adjustment*.

IV. Tool Description

Overlord is implemented as an extension, through a fork of *Clang*, a free and open-source C-family compiler tool chain. Having the implementation directly tied into a compiler is necessary as compilers optimise away the internal details of

the parsing as soon as possible in order to lessen the memory usage of the compilation process.

The version of our implementation, commit `e47090992`, with which we executed the experiment detailed in Section V, is available in source code form at *http:// github.com/ HoBoIs/ llvm-project/ tree/ clang-overload-debugger*. The user must self-compile the "`clang`" binary, which is the usual – but now extended – compiler. The details on how to accomplish this are platform-specific, and are available in the *LLVM Project*'s documentation.[1] Our extension of the compiler does not impose any additional system requirements.

During run-time, OVERLORD ties into and accepts callbacks from *Clang*'s "`Sema`" module, which is the major part of the compiler responsible for *semantic analysis*. OVERLORD is only an observer that synthesises information from the received data structures, but the behaviour of `Sema` is otherwise left intact.

A. Inspection

To **inspect** an overload situation in a source file, execute the `clang` compiler with the usual compilation flags needed for that file, and, in addition, specify the "`-Xclang -ovins-dump-opt=...`" flag. This *"overload inspection options"* argument takes various inputs, separated by commas, which are documented in the main *"README"* of the source repository. Some of the more notable options are: (1) a list of interesting source line(s), e.g., "**`2,4-8,15-16,23-42,2024`**"; (2) "**`CandName:fun`**" to filter for overload sets which contain "`fun`" as a *candidate*; (3) "**`{Show,Hide}NonViableCands`**" to request or suppress information about *non-viable candidates*; (4) "**`{Show,Hide}ImplicitConversions`**" to toggle diagnosing *implicit conversions*; (5) "**`{Show,Hide,Verbose}Compares`**" to toggle the verbosity of the reasoning about *which candidate is "better"*.

To exercise the tool with its simplest default configuration – including diagnosing every overload candidate set but excluding the contents of *header files*, – "`-Xclang -ovins-dump`" may be used instead.

The output is the detailed reasoning of the compiler's decisions, in the form of "warning-like" printouts, which is normally unavailable to users without our tool. For example, compiling Listing 1 by specifying the line number of `s5`'s construction, we obtain the answer to Section III's question (2) as shown in Listing 2.

B. Profiling

To **profile** the number and time spent for overload resolution, the project under test must compile all – or the selected – source files with the `clang` compiler containing OVERLORD, and the build system must be configured in a way that the compiler is passed the flags **`-Xclang -ovins-dump-opt=OnlyTime,PrintYAML`**, **`ShowImplicitConversions`**. Once configured, execute the projects' build normally.

The tool outputs a YAML file for each source file, containing one entry for each resolved function name and some meta-records with a special identifier, e.g., "`__TotalOverloadTime`". An entry describes (1) the number of "top-level" resolutions; (2) the time spent resolving "top-level";. Here, a resolution is ***"top-level"*** if it was performed not because it was needed to satisfy another resolution. To obtain project-level profiling information, the YAMLs can be reduced, entry-by-entry, trivially.

```
Result: Overloaded 'S::S': OR_Success with types [S]
    S s5(s4);                    // "Templated."

Best candidate: S::S(S&)
  Template parameters: [ T ::= S& ]
      ⇒ [ T&& ::= S&&& ≡ S& ]
    template <class T>
    S(T&&)       { puts("Templated."); }
  Conversions:
    - StandardConversion: [S ⟶ (S& ≡ T&&)]

Viable candidate: S::S(const S&)
    S(const S&) { puts("Copy.");       }
  Conversions:
    - StandardConversion: [S ⟶ const S&]

Comparing candidates: selected first
    (OVL_CC_BetterConversion)
  Conversions:
    - [S ⟶ (S& ≡ T&&)] ≥ [S ⟶ const S&]

Non-viable candidate: S::S()
  (OVL_Fail_TooManyArguments: needed 0, got 1)
    S()          { puts("Default.");    }
Non-viable candidate: S::S(S&&)
  (OVL_Fail_BadConversion: [S ⟶↯ S&&])
    S(S&&)       { puts("Move.");       }
```

Listing 2: Detailed reasoning for the selection of the better of two overload candidates of Listing 1. The output shows the exact decisions the compiler performed, complete with diagnostics-like hints to the source code fragments.

V. RESULTS

In addition to verifying the behaviour of OVERLORD on synthetic test examples – see Section III – we executed the *profiling* mode – as described in Section IV-B – as a whole-project analysis on a selection of C++ projects from various domain, scope, and size. We were interested in how much time is spent purely for semantic analysis – *Clang*'s `Sema` module – and within it, in connection with overload resolution. On top of measuring overloads in the "main body" of the source file, the "`ShowIncludes`" option was also passed, thus, for each translation unit, the entire contents were analysed.

We executed the measurement on a computer with Intel® Core™ i5-1145G7 CPU, 16 GiB DDR4-3200 MHz system memory, an SK-Hynix PC711 NVMe™ SSD storage device. The *Clang* project with OVERLORD's features was compiled in the optimised, "ReleaseWithAssertions" configuration. The analysed test projects were "compiled" with their unoptimised, "Debug" configuration.

[1] "Getting Started with the LLVM System", available at http://llvm.org/docs/GettingStarted.html, version `18.1.6`, accessed 2024-07-04.

TABLE I: Compile time and overload profiling result overview.

Project			Execution time					# *top-level*	# unique	% O-time
GitHub Repository	**Version**	**Commit**	**Compiler**	**Sema**	**(% C)**	**Overload**	**(% S)**	**resolutions**	**names**	**top 5 names**
EQMG/Acid	—	2ff8adee3	0:33:51	0:28:20	83.68%	0:09:36	33.92%	4 803 802	5 737	29.50%
bitcoin/bitcoin	v28.0	110183746	1:45:57	1:22:22	77.75%	0:32:02	38.88%	10 605 416	9 234	21.89%
contour-terminal/Contour	v0.5.1.7247	a7516ca56	0:04:53	0:03:34	73.10%	0:01:00	27.78%	363 504	1 258	53.13%
LibreOffice/core	24.8.3.2	48a6bac9e	14:58:22	12:21:06	82.49%	4:13:31	34.21%	129 119 402	69 525	27.53%
webmproject/libwebm	1.0.0.31	6745fd29e	0:00:16	0:00:11	68.63%	0:00:02	15.64%	31 442	642	29.21%
llvm/llvm-project	19.1.3	ab51eccf8	9:41:31	7:54:02	81.52%	2:59:36	37.89%	147 371 322	55 672	24.57%
OpenRCT2/OpenRCT2	v0.4.16	c1082a3d6	0:49:04	0:44:36	90.91%	0:12:45	28.60%	6 981 816	8 674	33.12%
protocolbuffers/protobuf	v28.3	5fda5abda	1:05:47	0:51:59	79.01%	0:16:19	31.39%	11 165 722	13 906	19.53%
qt/qtbase	v6.8.0	b839e9b36	1:36:24	1:01:25	63.72%	0:21:15	34.60%	9 361 528	14 616	26.79%
leethomason/tinyxml2	10.0.0	321ea883b	0:00:00.79	0:00:00.27	33.93%	0:00:00.04	14.81%	3 648	109	47.25%
apache/xerces-c	v3.3.0	31b4b3a06	0:02:42	0:01:51	68.61%	0:00:13	11.76%	316 854	1 227	45.33%

TABLE II: Details of the most striking contributors of overload resolution time, per analysed project. "ΣT" is the **total overall** time spent for the whole group. "**Sp.T**" is the **specific** time spent in the group: the average for *one* resolution.

(a) LibreOffice/core [avg. resolution time: 0.24 ms]

max. Σ time	#	ΣT (s)	Sp.T (ms)	*max. **Sp.** time*	#	ΣT (s)	Sp.T (ms)
basic_string_view	70 351	1 206.32	17.15	make_unordered_map	11	0.88	80.13
__is_implicitly_conible	613 828	1 143.51	1.86	frozen::unordered_map	18	0.65	36.31
__is_complete_or_unbounded	2 569 927	621.01	0.24	ItemPropertyMapType	11	0.37	33.83
=	2 626 397	618.64	0.24	erase_if	2 031	63.72	31.37
+	627 318	597.69	0.92	SwFormatCollConditions	673	20.66	30.70

(b) llvm/llvm-project [avg. resolution time: 0.15 ms]

max. Σ time	#	ΣT (s)	Sp.T (ms)	*max. **Sp.** time*	#	ΣT (s)	Sp.T (ms)
==	3 638 240	809.91	0.22	enumerate	224	5.39	24.08
__is_complete_or_unbounded	2 526 816	579.56	0.23	post_order	18	0.42	23.54
=	2 432 786	468.18	0.19	make_first_range	29	0.63	21.55
__is_implicitly_conible	296 700	441.84	1.49	__do_visit	61	1.28	20.95
make_pair	115 499	348.39	3.42	blocksOnly	16	0.32	19.98

For each project, we counted the total number of overload resolutions performed, the number of uniquely named groups that were overloaded, and several kinds of time spent for each group's resolution. We detail the quantitative results in Table I, including the processing times. We selected to sum the time taken by the top 5 named groups, and showcase that, almost everywhere, just the top 5 symbol names make up a significant portion of the total overload resolution time.

Now that we see that there is a sizeable majority of likely culprits contributing to the cost of overload resolution, it should be interesting to dig deeper. Indeed, we analysed the results from all 11 test projects, and detail the 5+5 worst offenders per project. In Table II, for the two largest projects we individually name the 5 overloaded symbol names from the two extreme ends of the gathered data. First, those that consumed the most **overall** time – these are the symbols that made up the percentages in Table I's rightmost column. Second, those that had the slowest **specific** resolution time: the average time it took the compiler to resolve their overload(s) **once**. We excluded the ones occurred less than 10 times, because there the measurement is more noisy (eg.: an unlucky push_back can trigger a big reallocation). For each group, we detail the number of resolutions performed for the group, and both the groups' *overall* and *specific* resolution time.

The functions with the most time spent overloading almost always consisted of some global operator overloads, e.g., "==". An other reoccurring deduction were intrinsics used by the standard library e.g.: `__is_complete_or_unbounded`. However, for these symbols, and the per-project overall measurement, the average time for a *single* overload resolution was surprisingly consistent, with the per-project average around 0.2-0.4 ms, and the specific time of the operator groups usually 0.5-1 ms.

Although the fact that operator symbols appeared in the top list was not unexpected, our data debunked the myth, that the "<<" is the most costly overload due to the *iostream* library. While "<<" is in the top 5 of 3 projects, a likely reason behind =='s more striking ubiquity is that it is as follows. "==" generally useful and required to be defined in almost every circumstance, and equality checks is one of the most often used predicates. All operator overloads for an operator – whether defined as global or member functions – will always belong to the **same** overload set. If it is reasonable to expect that "==" will be defined for most types, it means that every equality predicate must type check every other "==" that might appear, increasing the processing needs of every relevant resolution.

VI. Conclusion and Future work

In this paper we introduced OVERLORD, our tool extension of the free and open-source *Clang* C-family compiler. The tool has two main purposes: it helps to understand the outcome of C++ overload resolution step-by-step explaining how the compiler eventuated the actual result. On the other side, library developers can use the tool for profiling the compilation time spent to the overload resolution process, highlighting the most critical functions which can cause compilation bottlenecks. The tool is capable the handle the most modern C++ language features and applicable to large industrial projects, like the LLVM compiler infrastructure.

Future plans include handling upcoming C++ features introduced by new standard proposals and enhancing profiling granularity. Our intension is to upstreaming the tool to become a standard feature of the clang compiler family therefore it will be easily available for all C++ developers.

The tool is freely available in source code form at [8], and we also provide a *VirtualBox* image to facilitate testing [9].

References

[1] B. Stroustrup, "Operator overloading in C++," in *IFIP WG2.4 Conference on System Implementation Languages: Experience & Assessment*, 09 1984.

[2] ——, *The design and evolution of C++*, 1st ed. Addison-Wesley Professional, 03 1994.

[3] ISO and IEC, *Working Draft, Standard for Programming Language C++*. Geneva, Switzerland: International Organization for Standardization. [Online]. Available: https://www.iso.org/standard/83626.html

[4] J. F. Power and B. A. Malloy, "An approach for modeling the name lookup problem in the C++ programming language," in *2000 ACM Symposium on Applied Computing*, vol. 2. ACM, 03 2000, p. 792–796. [Online]. Available: http://doi.org/10.1145/338407.338564

[5] D. van de Voorde, N. M. Josuttis, and D. Gregor, *C++ Templates*, 2nd ed. Addison-Wesley Professional, 09 2017.

[6] R. Szalay, Á. Sinkovics, and Z. Porkoláb, "Practical heuristics to improve precision for erroneous function argument swapping detection in C and C++," *Journal of Systems and Software*, vol. 181, 07 2021. [Online]. Available: http://sciencedirect.com/science/article/pii/S016412122100145X

[7] G. A. Nagy and Z. Porkoláb, "Performance issues with implicit resolution in Scala," in *10th International Conference on Applied Informatics*. Eszterházy Károly University, 01 2017, p. 211–223. [Online]. Available: http://icai.uni-eszterhazy.hu/icai2017/uploads/papers/2017/final/ICAI.10.2017.211.pdf

[8] Botond Horváth, "Overload project repository," 11 2024. [Online]. Available: http://github.com/HoBoIs/llvm-project/tree/clang-overload-debugger

[9] Zoltán Porkoláb, "Overload test virtual machine," 12 2024. [Online]. Available: https://ikelte-my.sharepoint.com/:u:/g/personal/gsd_inf_elte_hu/EbDhxfyVd5xPgWhN-2plPdkBW7mgyrBCdADExfquQ5tRJQ?e=NELm9N

[10] C. Wang and D. Hou, "An empirical study of function overloading in C++," in *8th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 09 2008, pp. 47–56. [Online]. Available: http://ieeexplore.ieee.org/document/4637538

[11] Mozilla Foundation, "The Mozilla Project," 06 2002. [Online]. Available: http://mozillazine.org/articles/article2278.html

[12] A. Hück, C. Bischof, and J. Utke, "Checking C++ codes for compatibility with operator overloading," in *15th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 09 2015, pp. 91–100. [Online]. Available: http://ieeexplore.ieee.org/document/7335405

[13] A. Hück, J. Utke, and C. Bischof, "Source transformation of C++ codes for compatibility with operator overloading," *Procedia Computer Science*, vol. 80, pp. 1485–1496, 2016. [Online]. Available: http://sciencedirect.com/science/article/pii/S1877050916309553

[14] École polytechnique fédérale de Lausanne – Programming Methods Lab., "Scala 3 Reference: Changes in Implicit Resolution." [Online]. Available: http://docs.scala-lang.org/scala3/reference/changed-features/implicit-resolution.html

[15] M. A. Ellis and B. Stroustrup, *The annotated C++ reference manual*. Addison-Wesley Professional, 1990.

[16] P. J. Plauger, M. Lee, D. Musser, and A. A. Stepanov, *C++ Standard Template Library*, 1st ed. Pearson, 12 2000.

[17] G. Dos Reis and B. Stroustrup, "A formalism for C++," ISO/IEC JTC 1/SC 22/WG 21 — Open Papers, Tech. Rep., 10 2005. [Online]. Available: http://open-std.org/jtc1/sc22/WG21/docs/papers/2005/n1885.pdf

[18] ——, "A principled, complete, and efficient representation of C++," in *Joint Conference of ASCM 2009 and MACIS 2009*, 12 2009. [Online]. Available: http://stroustrup.com/macis09.pdf

[19] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*, 1st ed. O'Reilly Media, 12 2014.

[20] R. Szalay and Z. Porkoláb, "Visualising compiler-generated special member functions of C++ types," in *21st International Multiconference on Information Society: Collaboration, Software and Services in Information Society*, vol. G. University of Maribor, pp. 55–58. [Online]. Available: http://is.ijs.si/wp-content/uploads/2019/02/Zbornik-G.pdf