Proceedings of the 11th International Conference on Applied Informatics
Eger, Hungary, January 29–31, 2020, published at http://ceur-ws.org

Towards Decoupling Nullability Semantics from Indirect Access in Pointer Use*

Richárd Szalay

Eötvös Loránd University, Faculty of Informatics,
Department of Programming Languages and Compilers,
Pázmány Péter stny. 1/C., 1117 Budapest, Hungary,
szalayrichard@inf.elte.hu

Abstract

The special "null-pointer" (C, C++, Ada) or "null-reference" (Java, C#, Python) value for a pointer-like type is often used to indicate the lack of a meaningful result/data. Accessing a non-existing value is an unnatural operation, resulting in either unpredictable behaviour of the program or the raising of an exception. The usage of pointers often leads to a defensive design: it is expected of the programmer to pre-emptively guard against the null state of a pointer, or handle the resulting exception. Together with a code organisation principle to prefer "early returns", this defensive mechanism may result in variables in the local scope polluting the list of available symbols. These variables' existence does not pose a performance overhead at run time as virtually all compilers optimise the variable away by caching the loaded value. However, during code comprehension, these symbols remain visible, suggested by code completion tools which hinder understanding. Some programming languages offer "conditional dereference" operations: in C#, the `?.` operator propagates a null reference; in Haskell, the `Maybe` monad allows expressing such semantics. Modern C++ versions support expressing Maybe-like values with the `optional<T>` class template, but it encapsulates the value, not the indirect access. Adaptation of new language features or changing user-facing API is often met with business or technical challenges and is thus a slow process. In this paper, we discuss our investigation of the usage of pointer-like types (including iterators) for "nullability" semantics, not only for indirect access. We devised an automated analysis tool that marks potentially redundant pointer variables, lowering the number of visible local symbols. A post-refactoring view can show the landscape of the program where descent

*This work presented in this paper was supported by the European Union, co-financed by the European Social Fund in project EFOP-3.6.3-VEKOP-16-2017-00002.

in complex data structures (such as configuration maps) is expressed more concisely.

Keywords: C++ programming language, encapsulation, memory model, pointers, software design

1. Introduction

```
Configuration* Conf = GetGlobalConfiguration();
if (!Conf)
    return;

ConfigurationKey* SystemKind = Conf→getDataFor("SystemKind");
if (!SystemKind)
    throw InvalidConfiguration("config must contain 'SystemKind'!");

SystemType Sys = (SystemType)SystemKind→value;
switch (Sys) {
    case Platform1: /* ... */ break;
    /* ... */
}
```

Listing 1: Example of C++ program design preferring early returns when potentially null pointers are used. With minor transformations, similar code could detail Java or C#, too.

Several mainstream programming languages support indirect access to values in memory. In languages such as Java, C# or Python, it is through *references*, while other languages, such as C, C++ and Ada uses the term *pointers*, although there are no semantic differences between the two categories. An inherent property of pointers (or *indirect accessors*) is that there is the possibility to assign a *null* value to the pointer itself. This null value indicates that the pointer does not point to any valid value, and various programming languages have defined operating a null-pointer to be an unexpected operation. In managed or interpreted languages, such as Java, C#, Lua, Python, etc. a run time exception is raised at the *null dereference*. Lower level systems programming languages, such as C, C++ and Ada define accessing through a null to be an undefined operation, with potential – but not mandatory – run time support for defence against nulls. While method calls of an object through a `null` pointer is *undefined behaviour*[1] [8], the generated binary

[1] Code containing erroneous constructs marked as *undefined behaviour* by the standard allows the compiler to compile code that does not behave as the developer intended. The lingering concept of undefined behaviour allows a great set of optimisations to be made, as the compiler may pretend the developer did not intend to create undefined code. It is also commonly referred to as a *whatever-case*, but developers often end up sacrificing full portability for performance in case they need only support a specific platform.

will run a potential "prefix" of the method's code without any issue until the first member access – at which point an offset in memory from the `0x0` location should be read – is done.

Due to this, it is common in imperative programming languages with pointers to develop with a defensive design: the code at all points must either explicitly guard against the nulls, or specify in comments whether calling a function with a null argument is valid. Ada supports annotating the *access variable* with the `Not Null` tag, while Eiffel offers optional precondition checks. A common pattern together with the defensive design is to favour what is commonly referred to as "early `return`s". By *early returns*, we commonly refer to a technique where the unlikely or erroneous conditions break the flow – with any flow-control statement, not just `return` – of the program ahead, without the "normal" flow being indented visually. An example of early returns in the context of pointers potentially not referring to any reasonable value is shown in Listing 1.

The issue with having to resort to such coding style is that in many cases there is a potential to reach the useful business logic with several pointer variables visible in the local scope that are not used in just one dereference, often with a null check. For example, in Listing 1, the `switch` statement's condition could be written as `GetGlobalConfiguration()→getDataFor("SystemKind")→value`, were it not for the potential null value of the intermediate results. The increase of named symbols in a scope hinders development and code comprehension as development tools present additional potential suggestions.

In this paper, we discuss the problem related to `null` values, focused on the detection of potentially erroneous program points by an automated tool. In Section 2, related solutions from other languages and the theory are presented. Section 3 discusses the methodology behind finding bug-prone program sections. We measured open-source C and C++ projects for the prevalence of the problem, which is detailed in Section 4.

2. Related Work

Two potential solutions exist that allows hiding or side-stepping an explicit null check. The first such solution is mostly known from C# (version 6 and onwards) as the *null conditional operator* [11] or via the symbol "`?.`". The semantics of ?. is that the member access is only performed if the left-hand side is not `null`; otherwise, there is no calculation performed, and the result is `null` without evaluating the accessed member itself. The accessed member may be a data member or a method. Thus, the example in Listing 1 can be rewritten as `GetGlobalConfiguration?.getDataFor("SystemKind")?.value` — the result's type is the same as `value`'s type, and it is `null` if any intermediate step is `null`; otherwise, the result is precisely as if the same expression was written without the `?` tokens. Compared to explicitly guarding against the null in each step, a single-step or chained `?.` application has the benefit of concisely expressing a traversal of calls to obtain a result. It requires, however, a managed language context (where

```
getGlobalConfiguration :: Maybe Configuration
-- Configuration can not be "null" for this function!
getEntry :: String → Configuration → Maybe Entry
-- Entry can not be "null" either!
getValue :: Entry → Maybe Value

-- "Casting" omitted.
Sys = getGlobalConfiguration >>= (getEntry "SystemKind") >>= getValue
-- Type of 'Sys' is: Maybe Value
```

Listing 2: Concise writing of a traversal of a potentially nullable chain of function applications.

the null value is intercepted and coalesced before a further call could happen) and a language with reference semantics where virtually all variable the programmer could access has a null state. One downside of coalescing the null is that there is no way to distinguish if the obtained `null` was because the full expression without the `?`s evaluated to null, or because there was an intermediate step where the execution chain broke.

In Haskell, the `Maybe` monad can be used to express optional semantics. `Maybe` is a data type with two data constructors, `Just x` – for any value x – or `Nothing`, which indicates the lack of value. While the `?.` operator in C# works for virtually every type, the *Maybe* has to be "channelled through" the types of the functions involved. Functions taking a `Maybe x` must either un-box the `Just x` and be a partial function, or a total function must be defined that deals with the `Nothing` case. A polymorphic definition of the *bind operator* `(>>=)` can be used concisely express calculations where the `Nothing` case is defined to "just" pass the lack of value on. (>>=) is a function of type `Maybe a → (a → Maybe b) → Maybe b`, i.e. it takes a nullable of type `a` and applies a calculation on it (if it is not *null*) resulting in a potentially null of `b`.[2] An overview of the example of Listing 1 rewritten to Haskell's `Maybe` monad can be seen in Listing 2. The downside of this approach is that the point where the calculation failed cannot be easily found as the end result is a `Nothing`. However, developers may use pattern matching instead of the (>>=) operator, which allows for custom "error handling" codes. A famous example of a program written in Haskell that deals with the optionality of data and configuration is *Pandoc* [9].

Pattern matching has been suggested to be a language feature of C++ [12], with which inspecting values through a pointer can be transcoded to similar syntax as the `Just x` and `Nothing` cases of a function in Haskell. Outside pattern matching, the `optional<T>` class template can be used to wrap an object of type `T` into a "potentially" null context. The optional instance *owns* the encapsulated value,

[2]The actual type of (>>=) is more elaborate as it is designed too handle any `Monad` instance. It has been simplified for clarity.

and as such, there is no indirection in the access itself. The optional instance is convertible to `bool` and thus can be used in a conditional context, such as to facilitate early returns. Similar concerns arise with using `optional<T>` as would with using `Maybe t`, namely that the type has to be channelled through the types of every class and function involved.

Various other works detail finding [7, 21] or transforming [4, 5] programs which involve null pointers or references, with transformations involving automatically generating error handling semantics. Several related unsafe programming constructs have been identified by the authors in [1]. Our work is similar that it attempts to identify a potentially unsafe and development-hindering construct.

The modelled constructs in this paper fall into a subclause of *taint or garbage value* analysis, which is also often carried out by other means of static analysis, such as symbolic execution.

3. Approach

We created an automated tool based on the open-source LLVM/Clang Compiler Infrastructure's [23] static analysis framework, which can be used to understand how developers organise their code around potential null pointers in an automated fashion. The developed tool works by searching and marking *redundant pointers* in the code. Most compilers trivially support automatically warning for *unused variables*. A *redundant pointer* is a relaxation of the *unused variable* concept. While these marked variables are not unused in the most rigorous wording – there exists a usage point like a dereference and potentially a null check –, there is a chance for the variable to be elided.

Given the example in Listing 1, the idea behind redundant pointers can be seen. `Conf` and `SystemKind` are only used to store a memory address only to be dereferenced later or flow away on a comparison. Formally, we define potentially redundant pointers as follows.

Definition 3.1. Local variables of the function which are of a dereferenceable type (pointers, smart pointers, iterators, etc.) are **potentially redundant** if:

- initialised with a value at exactly one point
- used at most once in a "early flow" branch (such as an early `return` or `throw`) that only range-checks the pointer's value
- the pointer is passed as an argument or dereferenced exactly once in its lifetime

We refer to the "early flow branch" in the middle bullet-point as *guards*.

3.1. Rewriting single occurrences of variables

Our first goal is to check whether the existence of such local variables is justified. Abstract examples for the matched source code fragments can be seen in Listing 3.

```
struct T { int i; T* next; };
T * tp1 = malloc(sizeof(T)), * tp2 = init2(), * tp3 = init3();
free(tp1);      // Single argument passing as usage point.

if (!tp2)       // Single conditional check.
    return -1; // Flow breaking statement.
printf("%f", sqrt(tp2→i));
                // ∧~~~ Single dereference as usage point.

T* tnext = tp3→next;     // Single dereference as usage point.
T* tnext2 = tnext→next; // Ditto.
free(tnext2);            // Single passing of 'tnext2'.
```

Listing 3: Various cases of *potentially redundant pointers*.

```
free( malloc(sizeof(T)) );  // tp1 substituted.

int tp2i;
if (T* tp2 = init2(); (!tp2) || (tp2i = tp2→i, false))
    return -1;
printf("%f", sqrt(tp2i));   // tp2 substituted.

free(init3()→next→next);  // tp3, tnext and tnext2 substituted.
```

Listing 4: The examples of Listing 3 rewritten to omit the pointer.

There are various means to rewrite such potentially unnecessary pointers that depends on the language – C or C++ – and the standard – mainly whether C++17 is used or earlier – that applies to the project. The static analysis framework allows for suggesting code changes to the developer, and thus we wrote suggestions on potential rewrites. The previous example can be seen with the rewrites applied in Listing 4.

Certain constructs, such as pointer parameters of a function, loop variables that are pointers must be ignored by the matching rules as there are no reasonable ways of removing the variable while also keeping the semantics of the project intact. Furthermore, the example for `tp2` in Listing 3 may only be reasonably rewritten in the newer C++17 and C++20 standards, and certain other restrictions – such as that the accessed member must be *default-constructible* and *assignable* (either copy or move) – also apply. The semantics, in this case, is slightly changed due to the default construction, so it is the responsibility of the developer with domain knowledge to decide whether the rewrite is sensible.

3.2. Dereference chains

Modelling single occurrences allows for building *chains* of pointer dereferences, which we used to identify code constructs similar to those solved in other languages by `operator?.`, `Maybe`, or `optional` (see Section 2).

Definition 3.2. A sequence of *potentially redundant pointers* each used in single dereferences that initialises another variable forms a **chain**. The last variable involved in a chain is not necessarily of dereferenceable type.

For clarity, chains of two are ignored as they are considered in the single occurrence case detailed in Section 3.1. An example of such a chain is `[tp3, tnext, tnext2]` in Listing 3, which can be rewritten step by step eliding the pointer variable. In case a chain contains a flow breaking statement, the rewrite has to happen with breaking semantics to a varying degree and changing the types involved to, e.g. an `optional`. The head element of a *chain* might be unusual in a way that it cannot be removed without changing the interface of the program element at hand. Such is the case for chains that begin with a loop variable or a function parameter.

Another solution could be the proposal and implementation of a `?.`-like operator to the language, which takes care of seemingly continuing the calculation when a "null dereference" is encountered. However, given both the lack of a managed execution environment of Java or C# and the lack of referential transparency of functional languages like Haskell, an operation similar to `?.` has to be carefully designed to fit well within the language framework offered by current standard C++. Assuming such a feature existed, the example in Listing 1 could be rewritten as `GetGlobalConfiguration?.getDataFor("SystemKind")?.value`.

Chains are built by the traversal of the connections between the single occurrence cases which usages are marked initialising a new variable from the result of the dereference.

4. Evaluation

We measured a selection of free and open-source projects of various scale and domain for both C and C++ programming language. The measurement results for the single occurrence case (see Section 3.1) is detailed in Table 1. It is surprising to see that virtually all of the cases where pointers (and for C++, other dereferenceable types) are used in these mature projects happen without any checks for the `null` value. This could be attributed to a multitude of reasons. First, the measurement only considers pointers that have exactly one (meaningful) usage point – it could be that pointers with multiple usage points are checked more rigorously against a null dereference. Furthermore, various other means of checking for the potential null value exists, such as *assertions*, and macros or short predicate functions that evaluate to a logical value but do not pass the received pointer-like value further. The conclusions we can draw from the results is that these pointers contain a hidden invariant to them, namely, that they will not point to null. This lessens the type

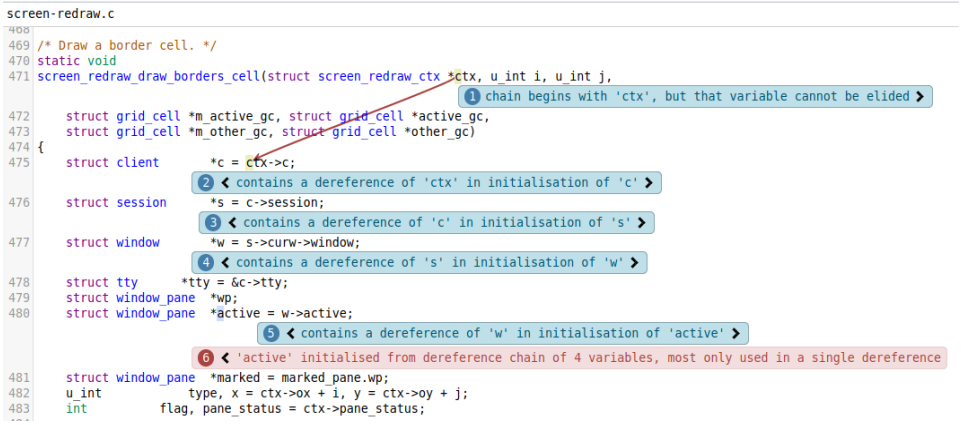

Figure 1: A chain of pointer dereferences initialising following variables that are further dereferenced but are not used anywhere in the function's body. While the local pointers `c`, `s`, and `w` could be elided, the head element of the chain, `ctx` is a function parameter which is not easily refactored. (Example is taken from TMux [10].)

safety of the system and should be rewritten to an Ada `Not Null`-like type if such would be available.

The measurement in Table 2 details the prevalence of chains (see Section 3.2) in the same set of analysed projects. The first clear result is that the number of guarded variables within a chain is virtually non-existent. The LLVM/Clang compiler infrastructure is an outlier with various well-guarded chains, and also shows multiple longer chains. Given the results, omitting the pointers could be a possibility to lessen the number of local variables in the function's scope. Another interesting result is that the amount of chains with a "trivial" (elidable) head is also low: almost all of the chains found have been identified with a *non-trivial head*. One such example is seen in Figure 1.

5. Future Work

The initial premise of this work was to show that redundant local pointer variables pollute the symbol list of the scope, potentially hindering development. We intend to extend the measurement tool in the future with also measuring how much percentage of the local symbols these pointers contribute, as in some cases these seemingly "unused" local variables might contribute to semantic understanding via their name [2].

More concise and compelling analysis methods, such as employing data-flow and path-sensitive analysis, might also lead to the discovery of further cases and potential refactoring efforts. As of writing this paper, data-flow information is not readily available for static analysis tools within the LLVM/Clang tool-chain.

While this work focused primarily on C and C++ projects, a similar problem with potentially elidable local references without a solution via language element exists in Java, which is an often-used application development language.

Lang.	Project	SLOC	Single occurrences				Expression		Dereference		Var-Init	
			No.	*G.*	Pointer	Object	No.	*G.*	No.	*G.*	No.	*G.*
C	curl [22]	146 756	***195***		195	-	81		104		10	
	git [24]	227 478	***820***		820	-	580		226		14	
	netdata [14]	79 984	***324***		324	-	271		45		8	
	PHP [16]	729 702	***1 151***		1 151	-	923		210		18	
	Postgres [17]	985 670	***3 237***		3 237	-	2 277		905		55	
	Redis [19]	133 995	***515***	*1*	515	-	393		103		19	*1*
	TMux [10]	48 456	***255***	*2*	255	-	142		83		30	*2*
C++	Bitcoin [13]	636 332	***351***	*9*	249	102	175		153		23	*9*
	guetzli [6]	8 029	***77***		76	1	75		2			
	LLVM/Clang [3]	5 052 050	***23 598***	*365*	17 858	5 740	17 726		4 298		1 574	*365*
	OpenCV [15]	1 009 759	***2 087***	*3*	1 835	252	1 752		294		41	*3*
	ProtoBuf [18]	268 466	***442***	*1*	382	60	280		152		10	*1*
	Tesseract [20]	157 506	***387***	*1*	384	3	228		102		57	*1*
	Xerces [25]	175 782	***763***	*5*	753	10	599		127		37	*5*

Table 1: The number of findings for single occurrences of potentially redundant pointers, detailed by finding type. `G.` columns indicate *guarded* usages. *Expression* usages are where the pointer variable is referred to in an expression, such as `free(ptr)`. *Dereference* usages are occurrences where the pointer is dereferenced to obtain a value. *Var-Init* is a special case of dereferencing counted separately, where the dereference is done in order to initialise another local variable. The table only contains cases that may be refactored within the current language rules.

Lang.	Project	Lengths				*Guarded variables*					Non-trivial head
		3	**4**	**5**	**6**	*0*	*1*	*2*	*3*	*4*	
C	curl [22]	58	2			60					59
	git [24]	34	3			35	2				37
	netdata [14]	28	4			30	2				32
	PHP [16]	23	2			24	1				25
	Postgres [17]	52	1			51	2				53
	Redis [19]	38	3			38	3				41
	TMux [10]	82	9	2		91	2				88
C++	Bitcoin [13]	1				1					1
	guetzli [6]										
	LLVM/Clang [3]	1 509	103	8	1	1 352	239	28	1	1	1 524
	OpenCV [15]	38				35	3				38
	ProtoBuf [18]	19				17	2				19
	Tesseract [20]	39				38	1				39
	Xerces [25]	102	3			99	6				101

Table 2: Details of the findings for potentially redundant *dereference chains* (see Section 3.2) for the project analysed. The findings are grouped by length of the chain, and for how many variables in each chain is *guarded* by a flow-breaking `if`. *Non-trivial head* indicates the number of chains in which the first element cannot be elided due to it being a function parameter, loop variable, etc.

References

[1] Baráth, Á., Porkoláb, Z.: *Towards Safer Programming Language Constructs*, Studia Universitatis Babeș-Bolyai, Informatica LX.1 (June 2015), pp. 19–34, ISSN: 2065-9601, URL: http://cs.ubbcluj.ro/~studia-i/contents/2015-1/02-BarathPorkolab.pdf.

[2] Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: *Exploring the Influence of Identifier Names on Code Quality: An Empirical Study*, in: 2010 14th European Conference on Software Maintenance and Reengineering, Mar. 2010, pp. 156–165, DOI: 10.1109/CSMR.2010.27.

[3] *Clang: a C language family frontend for the LLVM Compiler Infrastructure*, online, http://clang.llvm.org, version 9.0, accessed 2019-12-30, The LLVM Foundation, 2001-.

[4] Dobolyi, K., Weimer, W.: *Changing Java's Semantics for Handling Null Pointer Exceptions*, in: 2008 19th International Symposium on Software Reliability Engineering (ISSRE), Nov. 2008, pp. 47–56, DOI: 10.1109/ISSRE.2008.59.

[5] Durieux, T., Cornu, B., Seinturier, L., Monperrus, M.: *Dynamic patch generation for null pointer exceptions using metaprogramming*, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb. 2017, pp. 349–358, DOI: 10.1109/SANER.2017.7884635.

[6] *guetzli*, online, http://github.com/google/guetzli, version 1.0.1, accessed 2019-12-30, Google, Inc., 2016.

[7] Hovemeyer, D., Pugh, W.: *Finding More Null Pointer Bugs, but Not Too Many*, in: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '07, San Diego, California, USA: Association for Computing Machinery, 2007, pp. 9–14, ISBN: 9781595935953, DOI: 10.1145/1251535.1251537, URL: https://doi.org/10.1145/1251535.1251537.

[8] ISO/IEC JTC 1/SC 22: *ISO/IEC 14882:2017 Information technology — Programming languages — C++, version 17 (C++17)*, Geneva, Switzerland: International Organization for Standardization, Dec. 2017, p. 1605, URL: http://iso.org/standard/68564.html.

[9] MacFarlane, J.: *Pandoc: the Universal Document Converter*, online, http://pandoc.org, accessed 2020-01-19, 2006.

[10] Marriott, N. et al.: *tmux*, online, http://github.com/tmux/tmux, version 3.0, accessed 2019-12-30, 2007-.

[11] Microsoft Inc.: *Null-conditional operators ?. and ?[]*, accessed 2020-01-18, Sept. 2019, URL: http://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators#null-conditional-operators--and-.

[12] Murzin, S., Park, M., Sankel, D., Sarginson, D.: *Pattern Matching*, online, http://open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1371r0.pdf, accessed 2020-01-19, Jan. 2019.

[13] Nakamoto, S., The Bitcoin Core Developers, et al.: *Bitcoin*, online, http://bitcoincore.org, version 0.19.0.1, accessed 2019-12-30, 2009-.

[14] *Netdata*, online, http://my-netdata.io, version 1.19.0, accessed 2019-12-30, Netdata Corporation, 2013-.

[15] *OpenCV*, online, http://opencv.org, version 4.2.0, accessed 2019-12-30, Xperience AI, 2019-.

[16] *PHP: Hypertext Preprocessor*, online, http://php.net, php-src version 7.4.1, accessed 2019-12-30, The PHP Group, 1999-.

[17] *PostgreSQL*, online, http://postgresql.org, version 12.1, accessed 2019-12-30, The PostgreSQL Global Development Group, 1996-.

[18] *Protocol Buffers*, online, http://developers.google.com/protocol-buffers/, version 3.11.2, accessed 2019-12-30, Google, Inc., 2008-.

[19] Sanfilippo, S. et al.: *Redis*, online, http://redis.io, version 5.0.7, accessed 2019-12-30, 2006-.

[20] Smith, R., Google, I., et al.: *Tesseract OCR Engine*, online, http://github.com/tesseract-ocr/tesseract, version 4.1.0, accessed 2019-12-30, 2006-.

[21] Spoto, F.: *Precise null-pointer analysis*, Software & Systems Modeling 10.2 (2011), pp. 219–252, issn: 1619-1374, doi: 10.1007/s10270-009-0132-5, url: https://doi.org/10.1007/s10270-009-0132-5.

[22] Stenberg, D. et al.: *curl*, online, http://curl.haxx.se, version 7.67.0, accessed 2019-12-30, 1996-.

[23] The LLVM Foundation: *LLVM/Clang: C-language family frontend for the LLVM Compiler Infrastructure Project*, online, http://clang.llvm.org, accessed 2020-01-19, 2007.

[24] Torvalds, L. et al.: *git*, online, http://git-scm.org, version 2.24.1, accessed 2019-12-30, 2005-.

[25] *Xerces C++*, online, http://xerces.apache.org, version 3.2.2, accessed 2019-12-30, The Apache Software Foundation, 1999.