

Determining Method-Call Sequences for Object Creation in C++

Thomas Bach 
Heidelberg University

thomas.bach@informatik.uni-heidelberg.de

Ralf Pannemans
SAP SE

ralf.pannemans@sap.com

Artur Andrzejak 
Heidelberg University

artur.andrzejak@informatik.uni-heidelberg.de

Abstract—Unit tests in object-oriented programming languages must instantiate objects as an essential part of their set-up. Finding feasible method-call sequences for object creation and selecting a most desirable sequence can be a time-consuming challenge for developers in large C++ projects. This is caused by the intricacies of the C++ language, complexity of recursive object creation, and a large number of alternatives.

We confirm the significance of the problem by analysis of 7 large C++ projects and a survey with 143 practitioners. We then design an approach for recommending method-call sequences for object creation that align with criteria gathered by the survey. Our approach exploits accurate and efficient compiler-based source code analysis to build an object dependency graph that is processed by a divide-and-conquer algorithm.

An evaluation on a large industrial project shows that our tool find solutions that require in 99% of 1104 cases identical or fewer objects compared to manually crafted solutions. Developer feedback and manual analysis confirm these results. Moreover, solutions found by our tool require up to 6 times fewer objects on average compared to approaches from prior work.

Index Terms—object creation, testing, c++

I. INTRODUCTION

Object-oriented programming languages are widely used today [1–3]. Unit testing in such languages requires to initialize both objects under test and objects used as parameters. This is typically achieved by appropriate sequences of method-calls to create and mutate objects. In programming languages like C++ with a rather strict type system, it can already be challenging to find suitable method-call sequences to only *create* an object. Private constructors can require to detect non-obvious alternatives to create an object. Even more, creating an object might require recursive creation of additional objects and each of them can have a variety of alternatives for creation.

In such complex cases, developers must analyze the dependency hierarchy for a targeted object to (i) find feasible method-call sequences (in short *sequences*) for creating it, and then (ii) select a sequence they consider ‘optimal’. Focusing only on instantiating objects, we call the challenges (i) and (ii) the *object creation problem* (OCP).

In the context of unit testing, a wide range of research work proposed techniques for generating method-call sequences to set-up objects into a desired state [4–11]. In addition to these *sequence generation* methods, there are also *direct construction* methods [12]. These work focus on achieving a desired object state, assuming that finding

code for object creation is trivial. However, through our discussions with developers of SAP HANA, a large database system written in C++ [13], we noticed a high development effort for solving the OCP. The developers reported that finding and implementing object creation code as a part of unit test writing consumes a considerable amount of time. Interestingly, setting up an object state after its creation was typically considered a simpler task. We attribute the neglecting of the OCP in the literature to the relatively small project sizes that are used in evaluations and to the prevalence of other programming languages such as Java or C# in these previous work.

In this work, we first study the significance of the OCP and characteristics of suitable solutions by analyzing the code of large C++ projects and by conducting a survey with practitioners. Based on these results, we propose and evaluate an approach that suggests sequences of method-calls for object creation in C++ according to desired criteria. Our contributions are in detail:

- Confirming the significance for the OCP in 7 large C++ projects via static code analysis, and via survey results from 143 professional developers.
- Characterizing preferences of developers for object creation code based on the above-mentioned survey.
- An approach to suggest method-call sequences for object creation that considers the identified preferences.
- An evaluation on 7 large C++ projects analyzing the effectiveness and demonstrating the improvements compared to previous work.

This work is organized as follows. Section II motivates, Section III describes the survey, and Section IV shows our approach. Section V contains the evaluation, in Section VI we discuss the threats to validity, and in Section VII we outline related work. Section VIII provides the conclusions.

II. MOTIVATION

We show by examples how typical patterns of C++ source code can complicate the creation of objects.

Constructors: In Listing 1 we can create an object of type `CA` by calling the public constructor. We can instantiate `CB` by calling the public default constructor, which is implicitly generated [14, Clause 15]. Hence, a developer must know the definitions for implicitly-defined C++ default constructors in the C++ standard [14], which can be non-trivial in some cases. `CC` shows such a case. The object cannot be created with normal language constructs.

Listing 1: Three simple classes.

```

1 class CA {
2 public:
3     int a;
4     CA(int a)
5     :a(a){
6     /*...*/
7 }
8 };
9 struct CB {
10 int a;
11 };
12 class CC {
13 int& a;
14 public:
15     int id;
16 };

```

Listing 2: Object creation and inheritance.

```

1 class Base{
2 protected:
3     Base(int m) { /*...*/ };
4 struct Derived : public Base{
5     Derived(int n) { /*...*/ };
6     Derived(CC& c) { /*...*/ };

```

Listing 3: Object creation via factory pattern.

```

1 class P {
2 friend class PF; // a friend of P
3 P(int id) { /*...*/ };
4 struct PF {
5     PF(int id) : id(id) {}
6     PF(CC& c) : id(c.id){ /*...*/ }
7     unique_ptr<P> createP() {
8         return unique_ptr<P>(new P(7));
9     };

```

CC has no default constructor (the member `a` is a reference), and list-initialization is not possible (`a` is private) [14].

Derived Class: In Listing 2, an object of type `Base` cannot be created directly due to the non-public visibility of the constructor and absence of a default constructor [14]. However, inheritance allows us to use `Derived` for `Base`. We have two options to create an object of type `Derived`. We might select the `Derived(int)` constructor because the second constructor depends on additional objects.

Factory Pattern: In Listing 3, we are unable to create an object of type `P` directly as the only constructor is private. However, the class `P` declares `PF` as a friend. Therefore, the private constructor of `P` can be called from `PF`. Hence, to create an object of type `P`, we must discover the friend relationship to `PF`, identify and call `PF::createP()`, and recognize that the return type, a smart pointer, provides access to the desired object.

Conclusions: Object creation can be a difficult task even for short C++ programs. Large projects can have recursive object dependencies with several alternative options at each recursion level, resulting in a huge search space. Therefore, creating objects can be a time consuming task for developers in large projects.

III. COLLECTING DATA FROM USERS

We describe here the conducted interviews and a survey.

Exploratory Interviews: We conducted the following experiment with 3 developers from SAP for 90 min each. In the first half, we observed the developers at their work while they create unit tests. We noted their steps, and we qualitatively assessed the amount of time required by each step. In the second half, we interviewed the developers whether our observations were correct. Based on this data we created a list of their distinct activities with associated (relative) time requirements. The survey described below uses this list. We omit further descriptions of the interviews because they were only explorative while the survey provides the empirical foundation for our work.

Survey: We designed an electronic survey [15] where we ask the developers to rate the effort of different steps for unit test creation and to rate the importance of

different criteria to select among different options for object creation. We then conducted a trial run with 10 developers of SAP. Based on the results of this trial run and discussions with the participants we selected the specific formulations of the questions and the scale of the rating. For example, the trial group preferred the $-3 \dots +3$ rating scale over an initially proposed relative ranking.

The survey contains two questions shown in Table I. Each question has multiple items with 7-item Likert scales and a free text box for additional comments [15]. Table I shows all items of the second question based on the experience of our industry partner and related literature [16, 6]. The first question has as items 10 steps of the test creation process that we derived from information gained by the interviews: 1) understanding of the source code, 2) necessary refactoring of the code to make it testable, 3) conceiving test cases, i.e., thinking about possible input data and test oracles, 4) object creation/instantiation, 5) object state preparation, 6) mock creation, 7) writing test code (including test framework/build system code), 8) refactoring of the test code, 9) compilation/linking of test code, and 10) executing and testing test code.

Survey Participants: We target professional C++ developers. SAP sent the survey to 1185 recipients across multiple global C++ development teams. We assume the most participants are from Germany, North America, and Asia. Due to concerns related to European privacy laws, we have no details about cultural and experience distribution.

Results: We received 143 responses (participation rate of 12%). Not all recipients rated all items, therefore the number of ratings varies between 116 to 133. For the first free text box, we found 15 texts, and 6 for the second.

Table I presents the results. For the first question we ordered the 10 items (the steps of a test creation) by the mean of the responses. This yielded the following ranking of the items in descending order: 2, 6, 1, 4, 3, 7, 5, 8, 9, 10. The step ‘object creation’ is the fourth highest rated item. Higher rated items are (with corresponding arithmetic mean): ‘code refactoring’ (1.75), ‘mock creation’ (1.19), and ‘understanding of the source code’ (0.88).

For the second question, the highest ranked criterion is ‘minimal dependencies’, whereas ‘first working solution’ is ranked lowest. Interestingly, even though mutability simplifies the state preparation, the criteria ‘objects should be mutable’ is ranked relatively low.

The second free text box contained mostly opinions about testing in general. Two valuable remarks are that multiple product lines require additional effort and objects that change the global state should be avoided.

Interview and Survey Results

Professional developers in a large C++ project consider (a) time effort for implementing object creation as high, and (b) the minimal amount of dependent objects as the most important criterion for selecting a method-call sequence for object creation.

Table I: Survey questions and results. Column n indicates how many participants have rated the corresponding item.








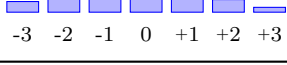
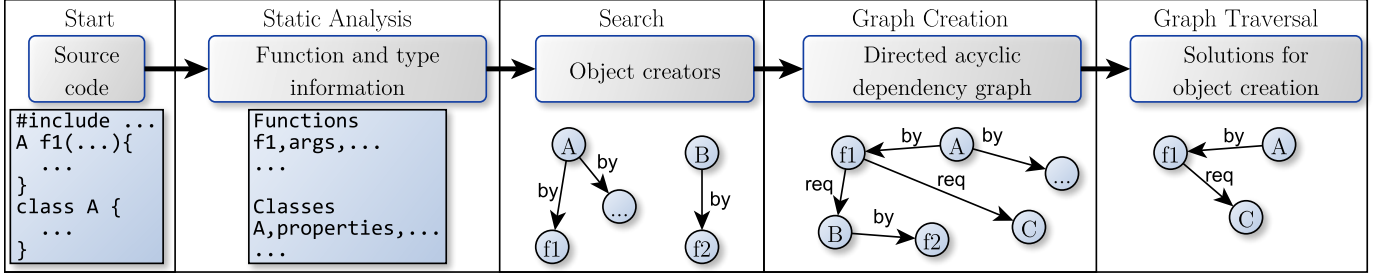
Items	Rating	Mean	Median	n
Question 1: Which aspects of implementing unit tests in SAP HANA require considerable time effort?				
4) Object creation/instantiation for source code under test.		0.83	1	133
	low time effort -3 -2 -1 0 +1 +2 +3 high time effort			
Question 2: If there are multiple options for object creation within a unit test (e.g., several constructors, factory methods), which criteria are important to select one option?				
1) The amount of dependent objects should be minimal, e.g., constructors with fewer additional object-type arguments are preferred.		1.73	2	120
2) The state of objects should be as mutable as possible to modify the objects during tests.		0.40	1	119
3) The object is created in the same way at other places in the productive source code.		1.09	1	120
4) The object is created in the same way at other places in the test code.		0.60	1	119
5) The objects should be related to the code under test, i.e., the distance between code under test and source code for the object should be minimal.		1.48	2	119
6) The objects should not be complex, i.e., metrics such as the cyclomatic complexity (understood as the complexities of the methods) or coupling should be minimal.		1.15	2	119
7) It should work at all, i.e., the first working solution is good enough.		-0.13	0	116
	not important -3 -2 -1 0 +1 +2 +3 very important			

Figure 1: The phases of our approach.

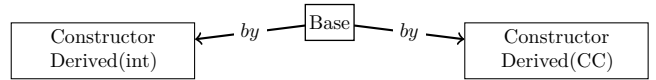


IV. APPROACH

Fig. 1 provides an overview of our approach to solve the *object creation problem* (OCP). The OCP asks to find a method-call sequence which instantiates an object of a desired type T such that the sequence satisfies or optimizes given criteria. Here, we focus on the OCP and do not consider mutating the created object into a specific state.

An object in C++ is typically obtained by calling a constructor or a factory method. The OCP is easy to solve if such ‘creators’ have no parameters. Otherwise, we may have to instantiate multiple parameters of a creator and have to recursively solve the OCP for each of them. This process can unfold in different ways, yielding alternative method-call sequences, each able to instantiate an object of a desired type. Without explicitly enumerating all valid sequences, we search for one which optimizes certain criteria (see Section III). We call such a sequence a *solution*.

Figure 2: Top nodes of a dependency graph for Listing 2.

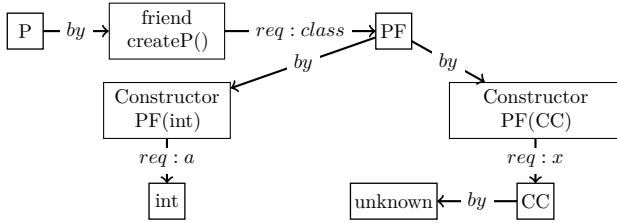


A. Object Creators and Dependency Graphs

For a given type, let S_c be the set of all functions which provide an instance (object) of this type, e.g., constructors or methods (see Section IV-B). We call such a function an (object) *creator* for a given type. In Fig. 2, type `PF` has two creators: constructors `PF(int)` and `PF(CC)`.

A *dependency graph* G for a desired type T is a directed acyclic graph with the following properties. All nodes of G correspond to either types or object creators. For a node v_t corresponding to a type t and each creator c of t , G has a node v_c (corresponding to c), and an edge $v_t \rightarrow v_c$ labelled with ‘by’. For each node v_c and each required parameter p of c , G has a node v_p corresponding to a type of the

Figure 3: Dependency graph for Listing 3.



parameter p , and an edge $v_c \rightarrow v_p$. We label such edges as ‘required’ (or ‘req’ for short) and specify what is required, e.g., the parameter name or the class instance. G might also have special nodes *unknown* linked from each type without any creator. Finally, there is a (root) node corresponding to the targeted type T . Given G , we define the *size* of G as the number of ‘type’-nodes. Fig. 3 shows a dependency graph for a targeted type P from Listing 3 with size 4.

Our approach has a (one-time) preprocessing phase to determine all creators for each type in the source code (Section IV-B). Given as input a desired type to be instantiated, we construct a corresponding dependency graph for this type (Section IV-C) and determine a solution via graph traversal described in Section IV-E.

B. Searching for Object Creators

We determine all creators for any type available in the source code by static analysis. To this purpose, we implemented a plugin for Clang, a C++ frontend for the LLVM compiler infrastructure [17]. The plugin extracts all data during compilation resulting in accurate data even in the case of multiple compilation stages where or complex preprocessor directives where the final source code for compilation is generated by other source code generation programs. The data contains all identified object creators and additional information for them such as order and types of parameters. In the following, we describe which language concepts are analyzed.

For a given type T , let S_c be the set of object creators for T . To simplify the presentation, we (a) use the term *class* also for *struct* and *union* [14, Clause 12], (b) shorten type T or class T to T , and (c) do not distinguish between references/pointers to an object and the object itself.

Constructors: We add to S_c all useful constructors for T and consider them technically as functions. A constructor is useful if it has *public* visibility, no attribute *deleted*, and is no copy or move constructor [14].

Inheritance: We add to S_c all accessible constructors for any subtype (multi-level, acyclic [14, Cl. 13]) of T .

Factory Method Pattern: We include in S_c all accessible static methods that return an object of type T . We ignore non-static methods. They are rarely useful because the underlying object must be created. They are only useful in the case of friends that we handle separately.

Friends: C++ allows a class to declare other functions or classes as *friends*. Friends can access non-public functions

and members of the class declaring the friend [14]. We add to S_c all methods that provide an instance of T and are marked as *friends*, or are defined in *friends*-classes of T .

Smart Pointers: A smart pointer is a data structure that wraps a pointer in C++ and provides additional functionality such as automatic memory management. We ‘shorten’ the indirection of smart pointers [14, 23.11] which seem to appear frequently in modern C++ code. E.g., for Listing 3, we use P instead of `unique_ptr<P>`. We detect such smart pointers by identifying the standard implementation cases and by providing a list of custom implementations of the corresponding project.

Output Parameter: Objects can be created by an *output parameter* pattern where a parameter value provided by the caller as input to a function is modified by the callee.

Listing 4 shows two examples of this pattern, *init1* and *init2*.

The first uses *call-by-reference*, the latter *call-by-sharing*. It is challenging to accurately detect such cases in general. Consider-

Listing 4: Output parameter.

```

void init1(T** o1){
    *o1 = new T(5);
}

void init2(O2& o2){
    o2.setT(new T(7));
}
  
```

ing all functions with adequate parameters would be misleading and requires further analysis. The decision problem whether a specific part of the code will be executed is rather complex and in general undecidable [18].

We use a heuristic that reports a function if any parameter is of type T and the body contains a constructor call for T . However, due to false positives, we do not include such results in S_c automatically but report them to developers for manual investigation.

Public Members: A second class U can contain a public member M of type T . We do not consider using M to be an intended way to create/access a desired object and therefore ignore it. Also, for creating T , U must use any of the detectable variants described before.

Static Casts: It is possible to create an instance of T by a static cast of another object. C++ allows such casts [14, 8.4] and also allows a memory copy without further type checks [14, 6.9, item 2]. However, such an approach could create incorrect states of objects in memory [19, 14] and is therefore not considered in S_c .

C. Constructing a Dependency Graph

Given the creators of all types, we can create a dependency graph for a type T . We start with a root node corresponding to T (node of level 0), and retrieve all creators of T . Each of them gives rise to a new ‘creator’-node v_c (level 1), and an edge between root and such a ‘creator’-node. Then for every parameter p of a ‘creator’-node v_c from level 1 we retrieve the type t of the parameter p , and create a (level 2) ‘type’-node v_t together with an edge $v_c \rightarrow v_t$ (see graph definition in Section IV-A). This process then repeats recursively for each newly created ‘type’-node (on the levels 2, 4, ...) until any of the following stopping criteria is met: 1) v_t has no creators. 2) v_t is fundamental, enum or function. 3) v_t is included in a white

Listing 5: ALG_o to find a solution for a given type. Function $minSize$ returns a non-empty argument with smaller size.

```

1 def ALG_o('creator'-Node vc)
2   Node result = copy of vc
3   for each parameter t of vc
4     Node solution = unknown # is empty
5     for each creator c of t
6       Node newSolution = ALG_o(c)
7       solution = minSize(solution, newSolution)
8     append solution to result
9   return result

```

list (a list that contains types for which we pre-define solutions, e.g., types that may have a large number of creators but are in fact simple to create). 4) v_t is included in a list of objects to mock (if a mock object should be used instead of the original object). 5) v_t is defined outside of the software project. 6) v_t is already processed (to prevent cycles). 7) The corresponding parameter is unnamed or has a default argument. 8) The number of creators is larger than a predefined threshold, e.g., 100 (such a large list may contain a suitable creator). 9) The recursion depth is larger than a predefined threshold, e.g., 5 (we do not expect any practical results at such depths).

D. Valid Method-Call Sequences

The dependency graph for type T allows finding all method-call sequences which instantiate T . There can be many such sequences. For example, in Listing 2, there are two sequences: one using the constructor `Derived(int)`, and another using the constructor `Derived(CC)`.

In general, a valid sequence corresponds to a subgraph H of a dependency graph with certain properties: 1) H contains the root node. 2) For every included 'type'-node v_t , H has exactly one child (a creator). 3) For each included 'creator'-node v_c , H contains all child nodes. 4) All leaves ('type'-nodes without descendants) correspond to types that can be created without requiring additional objects.

For practical use, we do not need to explicitly enumerate all valid sequences in a dependency graph. Instead, we can compute a single sequence that optimizes desired criteria, using the dependency graph as input.

E. Finding Solutions via Graph Traversal

We describe an algorithm that finds a solution for the OCP of a type T , i.e., a method-call sequence optimizing certain criteria. Following the results of the survey in Section III, our objective is fixed as the minimal number of dependencies, or 'size' of a sequence.

A solution is a subgraph of a dependency graph. Therefore, the size follows the definition given in Section IV-A and is the number of 'type'-nodes. For instance, Fig. 3 shows that the (unique) valid sequence for P has size 3, and the solution contains the types $\{P, PF, int\}$.

A solution is thus a sequence with a minimum size over all valid sequences. Listing 5 shows pseudocode of a divide-and-conquer algorithm to find a solution. ALG_o recursively finds a minimal subgraph and returns a solution

Table II: List projects for evaluation.

Project	#Obj. Types	#Functions	SLOC
SAP HANA 2018-11-24	735 194	4 210 541	11 065 382
Boost 1.66 [22]	30 548	53 528	4 392 925
CERN ROOT 6.13/08 [23]	100 705	730 507	3 417 362
Firefox 55.0.3 [24]	134 554	940 346	7 343 242
LLVM Clang 6 [25]	88 913	591 112	242 032
MySQL 8.0.11 [26]	54 360	199 692	3 791 989
ScummVM 2.0.0 [27]	13 527	148 350	1 830 628

(corresponding to a desired sequence) if it exists. Note that the algorithm can be easily modified to return multiple ranked recommendations.

ALG_o has the following properties: 1) It is a divide-and-conquer algorithm [20] by design. 2) We can use multiple threads for parallel execution. 3) We can re-use results for recursion. 4) We can use a branch and bound approach [20] to avoid descending recursion if a better solution is known. 5) The function `minSize` (line 7 in Listing 5) can be replaced to supports different optimization criteria.

V. EVALUATION

We investigate multiple research questions (RQ). First, we analyze the results of the search phase and characterize the studied projects (RQ1). Then, we study the existence of solutions (RQ2). Finally, we verify our solutions (RQ3), and compare our approach against related work (RQ4).

A. Evaluation Setup

We searched for large C++ software projects on GitHub, related literature and publicly available lists. We filtered the list of possible projects based on the following criteria. The project 1) uses C++ as the main programming language, 2) supports compilation with Clang, 3) has more than 10 000 different object types (it is 'large').

The resulting list contains 7 projects, see Table II. We measured the source lines of code (SLOC) with `cloc` [21]. More projects might fulfill these criteria, as we could not adapt some projects to compile with our plugin. Additionally, we pre-selected projects based on their expected number of types, i.e., we did not create statistics for all projects due to the required effort for the adaption to Clang compilation required by our Clang plugin.

We use a system with 4 processors, 160 cores with 2.10 GHz, and 1 TiB RAM. The Clang plugin increases the compilation time by a factor of 1.20 and generates 27 GiB of data for SAP HANA and 0.20 GiB to 8 GiB for the other projects. For practical reasons, SAP HANA requires parallel compilation. This would increase the intermediate data size to 4 TiB. Therefore, we implemented lock-free duplicate filtering to mitigate this issue. We consider these overheads acceptable for practical purposes. The execution time of our algorithm ALG_o (Listing 5) is below a second. This is considerably faster than a manual search, which can require more than 10 min per case according to our observations during the interviews with developers.

Table III: Distribution of default constructors. E.g., 94.56% of all classes in MySQL with 1-4 lines have a default constructor.

Project	Size Groups [LOC]			
	0..4	5..49	50..∞	All
SAP HANA	83.64	63.82	52.91	68.98
Boost	96.00	82.88	69.15	85.92
CERN ROOT	88.15	81.18	59.31	79.76
Firefox	93.49	79.02	81.32	83.24
LLVM Clang	87.26	79.62	56.55	77.55
MySQL	94.56	89.40	68.91	87.98
ScummVM	97.37	61.69	69.42	70.83

B. Search Phase and Project Characteristics

RQ1 What is the variety of types and the distributions of object creators found in the search phase?

For each project in Table II, we count the number of object types and their creators, the size of classes, and the presence of default constructors (*DC*).

Object Types, DC, and Class Sizes: We count each class as an object type. For class templates, we count explicit and implicit class template instantiations [14]. Table II presents the results. Table III shows for each class (a) the size $n = \text{lineEnd} - \text{lineStart} + 1$ grouped into small ($n < 5$), medium ($5 \leq n < 50$), and large ($50 \leq n$) as reported by our Clang plugin, and (b) whether it has a DC. Technically, a DC exists, if the existence is reported by the compiler and it is not marked as deleted [14].

Creators: For each object type T , we collect the set C_T of all creators and count $|C_T|$. In our dependency graph, $|C_T|$ is the number of nodes connected with a *by*-edge from T . For Fig. 3, **PF** has 2 creators. Fig. 4 presents the results.

Discussion: Fig. 4 shows that the search phase finds at least one creator for 93% to 99% of all object types. Among all projects, Clang has the highest percentage of empty results (7%). We conclude that the search phase provides reasonable results. A manual investigation of examples with empty solutions shows mainly cases where we were also unable to find solutions manually or where an object was used within a class-internal usage scenario. Further work is required to characterize such cases.

At least 6% to 20% of all object types and 16% to 38% of all large classes have more than 1 creator. As shown in Section II, it might be complex to find even 1 creator. These results confirm the relevance of the OCP.

The most frequent cardinality is 1 due to the presence of DC in small classes. Table III shows the distribution of DC in different size groups of classes. It is rather common for small classes to have a DC. Large classes often do not have default constructors, but multiple creators. This indicates that our approach is more effective for large classes.

Answer RQ1

We find creators for 93% to 99% of all object types. The most frequent result is a single creator. We find more than one creator for 6% to 20% of all object types.

Table IV: Results for all functions. Solved percentage of functions and the arithmetic means for the dependency graph sizes (*DG*), the solution sizes (*S*), and the number of arguments (*args*).

Project	%Solved	$\overline{ DG }$	$\overline{ S }$	$\overline{ args }$
SAP HANA	97.98	11.36	2.20	1.63
Boost	98.31	4.56	1.78	1.47
CERN ROOT	96.11	14.49	1.85	1.18
Firefox	97.16	15.16	1.95	1.39
LLVM Clang	94.74	18.96	2.03	1.28
MySQL	98.38	9.10	2.24	1.59
ScummVM	99.89	5.19	1.56	1.17

C. Existence of Solutions

RQ2 What is the fraction of functions where our approach can successfully find solutions for all arguments?

In practice, it is important to create all arguments of a function, which may require instantiating multiple different object types. Therefore, we switch our focus from objects to functions. In this section, we define a dependency graph (*DG*) of a function f as a union of dependency graphs for the types of each (required) parameter of f . Analogously, a valid sequence/solution for f is the union of the respective concepts over all parameters of f . In other words, we introduce an artificial root node, consider f as a creator for this root and apply our approach accordingly.

Functions: Our Clang plugin reports all functions generated by the compiler. This includes static functions, object member methods, lambdas, and each function template instantiation. Table II presents the results.

Size of Dependency Graphs (*DG*) and Solutions:

We apply our approach to each function and obtain a dependency graph DG and a solution S . Fig. 5 shows for all functions in each project the sizes of DG and S . Table IV presents the percentage of functions *%Solved* where our approach finds a solution and, over all functions, the average of the solution size $\overline{|S|}$, the graph size $\overline{|DG|}$, and the number of parameters $\overline{|args|}$.

Discussion: Table IV indicates that our approach finds full solutions to create all arguments for 94% to 99% of all functions. Fig. 5 shows that the solution size (Section IV-E) is typically rather low compared to the size of DG . The average solution size $\overline{|S|}$ is slightly larger than $\overline{|args|}$, which is expected. Unnamed arguments rarely occur in our data. Fig. 5 also indicates a rather large amount of functions with large DG sizes above 99. In some rare cases, the size is larger than 100 000. This aligns with our original motivation, that a manual inspection of the full space of possible solutions is either not practical or even not feasible in a reasonable amount of time. However, our search phase may collect object creators that would be discarded directly by a developer. Such cases artificially increase $\overline{|DG|}$.

We manually investigated unsolved functions. They contain in no particular order: (a) templates, (b) types defined but not implemented, (c) types provided by the operating system (d) types that were supposed to be non-constructable, and (e) types we found no way to manually

Figure 4: Histogram for number of object creators per type (x-axis in log-scale). E.g., for Boost, the search phase finds no creator in 1236 cases and exactly 1 in 27306 cases.

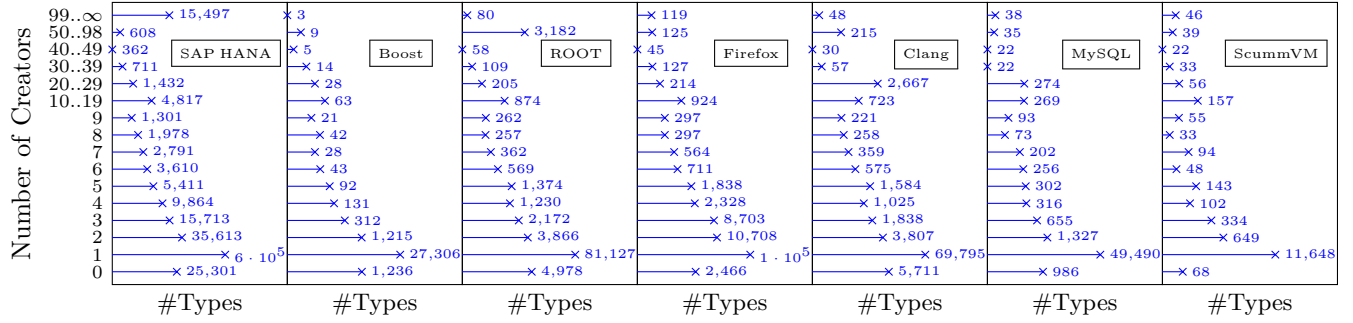
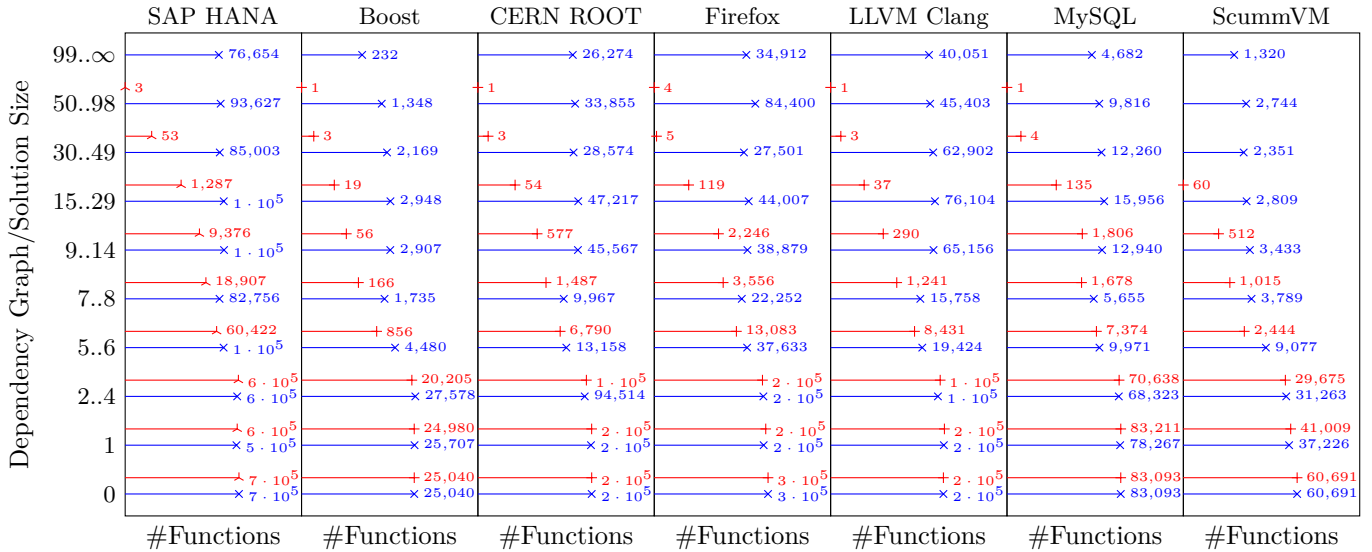


Figure 5: Dependency graph sizes — and solutions sizes — (x-axis in log-scale). E.g., Boost has 2169 functions with dependency graph sizes in range 30-49, and only 3 functions with solution sizes in this range.



construct them. We expect that fine-tuning the system with domain knowledge could improve (b), (c), and (d). A more sophisticated template analysis, which might be rather complex [28], may improve (a).

Answer RQ2

Our algorithm for object creation finds at least one solution to create all required objects for 94% to 99% of all functions in the evaluated projects.

D. Verification of Solutions

RQ3 How do the solutions found by our approach compare to manually found solutions with respect to the most important criteria identified?

We compare object creations by developers found in the source code of SAP HANA versus those automatically proposed based on preferences stated by developers in our user study. The comparison involves manual tasks, therefore we evaluate only a subset of all object creations. To collect these examples, we extend our static analysis to report all locations in the source code where an object is

created. We then filter them by a selection process and analyze the results. Additionally, we a) ask developers of SAP HANA to propose multiple problem instances and evaluate corresponding solutions, and b) manually investigate problem instances for the other projects.

Selection of Examples: We apply the following filtering steps: 1) SAP HANA consists of about 300 components that can be considered as own projects. We randomly select a set C_{50} of 50 components. 2) For each item in C_{50} , we collect the set OC of all source code locations where objects are created. 3) We only keep items in OC where all of the following conditions are true: file extension $\in \{cpp, cc, h, hpp, inl, inc, incl, hh, c\}$, source filename contains $['T/t]est$ (test code), object not in namespace $testing$ (test framework) nor std (standard library) nor X where X is an internal reimplementations of the standard library. 4) After filtering OC , we randomly select a set OC_{50} of up to 50 items. 5) We analyze each item in OC_{50} , and remove those that create artificial test objects such as a test fixture or a mock. This results in a final set of examples for each component. Table V reports statistics for each step.

Table V: Selectivity of the examples selection. We use the annotation (*min/max/mean*) for the statistics, e.g., (1/11/6) for {1, 5, 7, 11}.

Step	Statistics After the Step
(0) Start	3 539 879 object creations
(1) 50 random components	#files: (2/1 311/267)
(2) Object creations	661 886 object creations
(3) Filter step	113 081 object creations per component: (0/17 086/2 262)
(4) 50 random examples	19 components with 0 examples
(5) Remove test objects	1 104 object creations per component: (0/49/22)

Categorization: For each example, we apply our approach and generate a solution. We compare this solution to the manual object creation indicated by the existing source code. Table VI reports the results.

Out of 1 104 analyzed examples, the category *Shorter* contains 594 items, *Identical* 505 items, and *Longer* 5 items. Within *Shorter*, there are 113 cases where the solution used source code where the last change date is after the existing object creation, and in 481 cases before. Within *Identical*, there are 179 cases where the solution used source code where the last change date is after the existing object creation, and in 326 cases before.

Date Analysis: Source code changes could impact the retrospective analysis. Our approach could propose to use code that was not available for a manual solution. Utilizing the version control system to use the specific version would require recompilations and static analyzes with an estimated effort of 138d of execution time and 45 TiB of disk space. Due to limited resources, we instead extend our analysis to control for the described threat.

We use the version control system to calculate a date D_S for a solution, i.e., the last date when source code involved in a solution was modified, and identify the date D_M the manual solution was introduced. In Table VI, D_a reports number of cases for $D_S > D_M$ and D_b for $D_S \leq D_M$.

Manual Verification: Developers of SAP HANA proposed 10 recent problem instances, i.e., functions they wanted to call. One function requires only fundamental arguments, the others require at least one object, i.e., they represent complex scenarios. We apply our approach and ask the developers to evaluate the results. In 8 cases, they determine the results are identical to their own solutions. In 1 case, the result is better due to the correct identification of a default argument in a header file. In 1 case, our approach found no result. However, the developers revealed they also found no solution for this case and a solution probably does not exist. Hence, our approach found identical or better solutions in all cases.

For each of the other projects, we randomly select 10 object types from files with paths containing the string $[T/t]est$. We apply our approach and report the results as a tuple (smaller/identical/larger) that shows the comparison of solution sizes found by our approach compared to those found in the source code. ScummVM, ROOT, and MySQL:

Table VI: Solution sizes of ALG_o vs. manual solutions.

Category	n	% _{Total}	D_a	%	D_b	%
Total	1 104	100.00			N/A	
Shorter (S)	594	53.80	113	19.02	481	80.98
Identical (I)	505	45.74	179	35.45	326	64.55
S or I	1 099	99.55	292	26.57	807	73.43
Longer	5	0.45			N/A	

all (0/10/0). Boost: (1/9/0). The smaller case involves a custom smart pointer that would require special case handling and domain knowledge of the project. Firefox: (1/9/0). Clang: (2/8/0). One smaller case is within a test framework not controlled by the project.

Discussion: In 45.74% of the 1104 examples for SAP HANA, solutions found by our approach are identical to existing solutions and smaller in 53.80% of all cases. The 5 larger cases involve complex template metaprogramming [14] and interfaces with a high number of implementations. Still, developers might choose other solutions due to specific requirements. However, in the context of test creation, the functionality for objects not under test is typically not important. Hence, we assume correctness of our results in such scenarios. Even more, as indicated by our survey result in Table I, developers consider that the task ‘object state preparation’ requires less time effort. Therefore, we assume that this task might be simpler in large C++ projects compared to object creation.

The date analysis indicates that the impact of code changes for the retrospective analysis is low. For the category *identical*, the source code of the calculated solution must have been available at the time the manual solution was introduced. However, in 35.45% of all 505 cases, $D_S < D_M$. Hence, 36% is a threshold of expected cases. For the category *smaller*, 19.02% is below this threshold, confirming the initial statement.

Given the results of the date analysis and considering the results of the manual verification, we conclude that our approach is able to propose correct solutions for the object creation problem in practical cases.

Answer RQ3

In 99.55% of all 1 104 cases, our approach proposes solutions identical (45.74%) or smaller (53.80%) compared to existing solutions.

E. Comparison Against First-Working Approach

RQ4 How does our approach of a size-minimal solution compare against a first-working-solution approach?

Methodology: Related work uses the first working solution that is found, an approach that we identify by ALG_{fw} . For ALG_{fw} in comparison to ALG_o , the function $minSize$ (line 7 in Listing 5) is replaced by a check whether the subgraph is non-empty and, if this is the case, the loop is then aborted. We compare ALG_o against ALG_{fw} in terms of sizes of solutions for all functions. However, we

Table VII: Comparison between ALG_o and ALG_{fw} .

Project	$ \bar{S} $, all		$o \neq fw$ %	$ \bar{S} , \neq$	
	o	fw		o	fw
SAP HANA	3.10	6.81	42.07	4.45	13.26
Boost	2.57	3.90	31.80	3.13	7.31
ROOT	2.94	11.41	56.09	3.87	18.97
Firefox	2.88	5.94	44.69	3.67	10.51
Clang	2.62	11.64	60.17	2.96	17.94
MySQL	3.35	5.91	36.31	4.87	11.92
ScummVM	2.73	3.74	35.02	3.78	6.66

only investigate functions that require additional objects because all other functions have equal solutions. Additionally, we ignore functions where the return type is also one of the parameter types. These filtered functions represent 38% to 68% of all functions depending on the project. We do not report the number of solutions for ALG_o and ALG_{fw} , as they are indeed identical.

Results: Fig. 6 shows the histograms of solution sizes $|S|$ for each project and approach. Fig. 7 presents only the cases where the solution sizes are not equal. Table VII shows statistics for all cases. The execution time of ALG_{fw} is typically lower than ALG_o . However, the graph construction requires considerably more time compared to finding a solution that typically finishes within the fraction of a second. For example, the analysis for Clang shows the largest execution time with 0.15s per function on average. ALG_{fw} may produce different results depending on the order of the input. Given the large number of functions, we expect that this randomness is not a threat, and we did not further investigate different orders.

Discussion: Fig. 7 shows that size 1 does not occur for ALG_{fw} . The design of ALG_o guarantees $|ALG_o(f)| \leq |ALG_{fw}(f)|$. We removed all cases where $|ALG_o(f)| = |ALG_{fw}(f)|$. Hence, $0 < |ALG_o(f)| < |ALG_{fw}(f)|$ and therefore $1 < |ALG_{fw}(f)| \forall f$.

Over all projects, solutions based on ALG_o are on average by a factor 1.37 to 4.44 smaller for all functions or by a factor 1.76 to 6.07 smaller ignoring functions with identical solutions for both algorithms. Hence, in comparison to ALG_{fw} , ALG_o can considerably reduce the amount of objects and therefore decrease the complexity of solutions. Fig. 7 also shows that ALG_o effectively reduces cases with large solutions. Therefore, we conclude that our approach improves ALG_{fw} .

Answer RQ4

Solutions by ALG_o require up to 6 times fewer objects on average compared to a first-working-solution approach.

VI. THREATS TO VALIDITY

We discuss several threats to validity for our work.

1) *User Study:* Participants in the user study may not have the professional experience to answer the questions [29]. We reduce this threat by sending the survey to professional developers. However, we are unaware of

the number of participants without C++ experience. All participants are related to our industrial partner. However, we are not aware of any company policy that may influence our anonymous survey. With respect to diversity, the recipients are distributed worldwide and have different professional experience.

The user study might be ambiguous or the lists of items might be incomplete. We reduce this threat by a trial run.

2) *Reliability:* We collected a set of 7 projects for our evaluation. However, the composition did not follow a reproducible methodology, because we are unaware of a definitive list of large C++ projects. Due to the regulations of our industry partner, the implementation of our approach is not publicly available and an exact reimplementation of our approach may not be feasible. We carefully tested our implementation with an extensive test suite of collected C++ code examples and therefore expect that the conclusions are reproducible.

3) *Construct Validity:* Our evaluation contains a retrospective analysis. We are unaware of the reasons why a specific object creation option was selected in the past or whether such preferences have changed over time. A/B testing could mitigate this threat. However, it would require extensive resources to do such testing in large scale, therefore it was out of scope for our work.

We use the same set of projects in two aspect of our evaluation. We confirm the significance of the object creation problem and we analyze the effectiveness of our approach on the same 7 large C++ projects. Our approach may only be effective for projects where the object creation problem is an issue. We assume that our results generalize for other large projects that use an object-oriented programming language with a rather strict type system. However, our approach may not be interesting for small projects or dynamically typed languages.

4) *Internal Validity:* The search phase may find creators that are technically feasible, but practically not. Also, we may iterate uninteresting creators for the graph traversal. Thus, we may investigate cases that would be discharged directly by developers. This may produce more work for ALG_o but does not affect our conclusions.

5) *External Validity:* We assume that our approach can be generalized to other object-oriented programming languages with type information. However, in small projects, object creation may not be noticed as a problem. The user study results for time efforts may be specific to the rather strict C++ type system. The ranking of criteria provides guidance for other programming languages, too.

VII. RELATED WORK

Several other work on testing object-oriented programs either focus on the broader problem to generate a desirable object state [4, 11, 6, 30, 8, 9, 7, 5] or do not consider the object creation and, for example, capture objects during runtime [7]. Our work focuses on the specific part of object creation and does not aim to generate a desirable state.

Figure 6: Comparisons ALG_o (left) versus ALG_{fw} (right) for functions. The histograms show how often each solution size occurred.

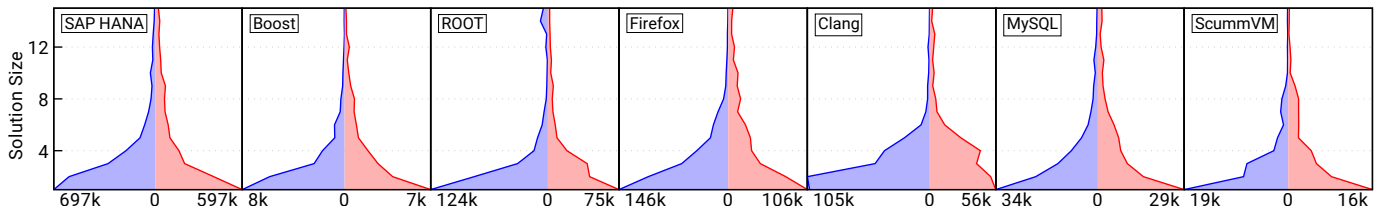
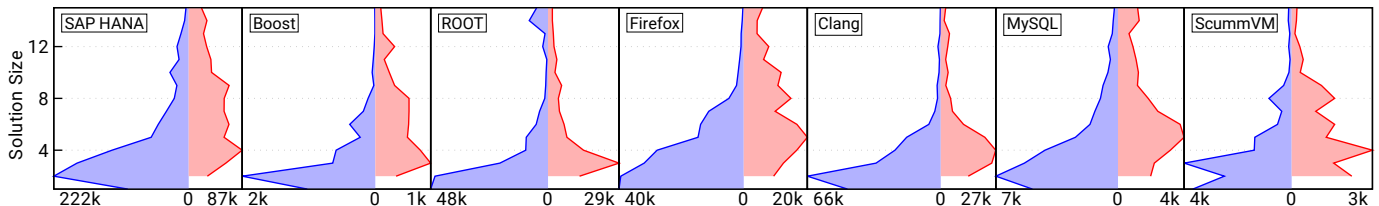


Figure 7: Presentation similar to Fig. 6. However, all cases where both solutions have equal sizes are ignored.



The work of Thummalapenta et al. [5] is closely related to our work. They also identify the problem of object creation as challenging and propose to use a keyword-based code search in method bodies within an intra-procedural analysis. However, they state that such analysis ‘is less precise than inter-procedural analysis’ and only used due to scalability reasons. Due to the keyword-based search, their approach does not require or use type information. The type detection by an inter-procedural analysis of our approach provides accurate results and is still fast.

Compared to API mining approaches [31], our approach can find more object creators and new solutions.

Several related work provide approaches for test generation in object-oriented programming languages [7, 32, 4, 11, 6]. Such approaches require a mechanism for object creation. They either (a) search for a constructor or generate the object with a general mechanism provided by the programming language [7, 32] or (b) recursively traverse the required dependencies for object creation until they find the first working solution [11, 6]. However, considering only the first working solution is undesirable in real-world projects according to the results of our survey. Cseppentő and Micskei provide further evaluation of test generation tools and their support regarding objects [33].

Previous work, where the evaluation targets comparatively small projects, recognize the challenges for the creation of complex objects [10, 11, 34].

The tools KLOVER [35, 36] and FSX [37, 38] automatically generate unit tests for large C++ projects. The examples shown for FSX contain a default constructor call, hence object creation seems to be supported. However, the approach remains unclear. Garg et al. target unit test generation in C++ with directed random test generation [39].

We are not aware of other studies on developers’ preferences for the optimization version of the object creation problem. We are unaware of existing tools for C++ or Java that respect such developer preferences.

VIII. CONCLUSIONS

The task of object creation in large C++ projects can be a time-consuming challenge for developers. Our approach automatically finds options for the object creation in more than 94% of all cases and solutions to create all required objects for 94% to 99% of all functions. Therefore, our approach can provide significant time reductions for developers. In addition, our approach finds solutions that better align with preferences of developers compared to manual solutions. Thus, using our approach can improve code quality. Finally, solutions found by our approach better align with preferences of developers compared to solutions found by a random approach used in related work.

In practice, developers could consider additional requirements for creating objects. Or even more, they might add new constructors to solve the problem. However, even in such cases, our approach can provide a list of alternative options and additional insights for *missing* options.

Our work addresses only object creation and does not focus on creating a desired state of an object. However, according to our survey results, developers state that the task of object state preparation requires less effort compared to the creation in the context of large projects. In addition, it may happen in the context of testing that an object instance is only required because a parameter demands it but the actual behavior may not be of interest.

While our work focuses on large C++ projects, we expect that the results generalize to other object-oriented programming languages with type information. We also expect that the results of our work allow other researchers to propose techniques with higher practical acceptance.

Future work on the connection between the creation of objects and their set-up into a desired state may provide additional benefits for automated tools and the manual work of practitioners. In addition, the accurate detection of output-parameters as described in Section IV-B might be an important task for C++ and C projects.

BIBLIOGRAPHY

- [1] TIOBE Company. 2020. *TIOBE Programming Community Index*. TIOBE Company. Retrieved 2020-01-10, archived by Internet Archive at <https://web.archive.org/web/20200110152937/https://www.tiobe.com/tiobe-index/> from <https://www.tiobe.com/tiobe-index/> Rank 1 to 5: Java, C, Python, C++, C#. (Cited from: I)
- [2] Stephen Cass. 2018. *The 2018 Top Programming Languages*. IEEE Spectrum. Retrieved 2020-01-10, archived by WebCite at <https://www.webcitation.org/76J2fuhpE> from <https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages> Rank 1 to 6: Python, C++, Java, C, C#, PHP. (Not cited.)
- [3] GitHub. 2018. *The Fifteen Most Popular Languages on GitHub by Opened Pull Request*. GitHub Inc. Retrieved 2020-01-10, archived by WebCite at <https://www.webcitation.org/76J2LOTd3> from <https://octoverse.github.com/2018/> Rank 1 to 6: Javascript, Python, Java, Ruby, PHP, C++. (Cited from: I)
- [4] Carlos Pacheco, Shuvendu K. Lahiri, Michael Dean Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, Washington, DC, USA, 75–84. ISBN 0769528287 DOI 10.1109/ICSE.2007.37 (Cited from: I and VII)
- [5] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (Amsterdam, The Netherlands) (ESEC/FSE 2009)*. ACM, New York, NY, USA, 193–202. ISBN 9781605580012 DOI 10.1145/1595696.1595725 (Cited from: VII)
- [6] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (Szeged, Hungary) (ESEC/FSE 2011)*. ACM, New York, NY, USA, 416–419. ISBN 9781450304436 DOI 10.1145/2025113.2025179 (Cited from: III and VII)
- [7] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: Object Capture-Based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (Trento, Italy) (ISSTA 2010)*. ACM, New York, NY, USA, 159–170. ISBN 9781605588230 DOI 10.1145/1831708.1831729 (Cited from: VII)
- [8] Sai Zhang, David Saff, Yingyi Bu, and Michael Dean Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (Toronto, Ontario, Canada) (ISSTA 2011)*. ACM, New York, NY, USA, 353–363. ISBN 9781450305624 DOI 10.1145/2001420.2001463 (Cited from: VII)
- [9] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramlar. 2015. GRT: Program-Analysis-Guided Random Testing. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE 2015)*. IEEE Computer Society, Washington, DC, USA, 212–223. ISBN 9781509000258 DOI 10.1109/ASE.2015.49 (Cited from: VII)
- [10] Sina Shamshiri, Rene Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE 2015)*. IEEE Computer Society, Washington, DC, USA, 201–211. ISBN 9781509000258 DOI 10.1109/ASE.2015.86 (Cited from: VII)
- [11] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs (Prato, Italy) (TAP 2008)*. Springer-Verlag, Berlin, Heidelberg, 134–153. ISBN 9783540791232 (Cited from: I and VII)
- [12] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. *ACM SIGSOFT Software Engineering Notes* 27, 4 (July 2002), 123–133. DOI 10.1145/566171.566191 (Cited from: I)
- [13] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Record* 40, 4 (Jan. 2012), 45–51. DOI 10.1145/2094114.2094126 (Cited from: I)
- [14] ISO. 2017. *Programming Languages – C++*. Technical Report ISO/IEC 14882:2017. International Organization for Standardization, Geneva, Switzerland. (Cited from: II, IV-B, IV-B, IV-B, IV-B, IV-B, IV-B, IV-B, IV-B, V-B, V-B, and V-D)
- [15] Jon A. Krosnick and Stanley Presser. 2009. Question and Questionnaire Design. In *Handbook of Survey Research* (second ed.), Peter V. Marsden and James D. Wright (Eds.). Emerald Group Publishing, Bingley, UK, Chapter 9, 439–455. (Cited from: III)
- [16] Davide Spadini, Mauricio Aniche, Magiel Bruntink, and Alberto Bacchelli. 2017. To Mock or Not to Mock? An Empirical Study on Mocking Practices. In *Proceedings of the 14th International Conference on Mining Software Repositories (Buenos Aires, Argentina) (MSR 2017)*. IEEE Press, Piscataway, NJ, USA, 402–412. ISBN 9781538615447 DOI 10.1109/MSR.2017.61 (Cited from: III)
- [17] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, Washington, DC, USA, 75–86. DOI 10.1109/CGO.2004.1281665 (Cited from: IV-B)
- [18] Martin Davis. 1983. *Computability and Unsolvability*. Dover Publications, Mineola, New York, USA. ISBN 9780486151069 (Cited from: IV-B)
- [19] Lei Ma, Cheng Zhang, Bing Yu, and Hiroyuki Sato. 2017. An Empirical Study on the Effects of Code Visibility on Program Testability. *Software Quality Journal* 25, 3 (Sept. 2017), 951–978. DOI 10.1007/s11219-016-9340-8 (Cited from: IV-B)
- [20] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (third ed.). The MIT Press, Cambridge, Massachusetts London, England. ISBN 9780262033848 (Cited from: I and 4)
- [21] Al Danial. 2020. *cloc – Count Lines of Code*. Retrieved 2020-01-10, archived by WebCite at <http://www.webcitation.org/76J5fFUlo> from <https://github.com/AIDanial/cloc> (Cited from: V-A)
- [22] Boost development team. 2018. *boost 1.66.0*. Retrieved 2020-01-10 from https://dl.bintray.com/boostorg/release/1.66.0/source/boost_1_66_0.tar.gz (Cited from: II)
- [23] CERN ROOT development team. 2018. *CERN ROOT 6.13/08*. CERN. Retrieved 2020-01-10 from <https://>

- root.cern.ch/download/root_v6.13.08.source.tar.gz (Cited from: II)
- [24] Firefox development team. 2018. *Firefox 55.0.3*. Mozilla. Retrieved 2020-01-10 from <https://archive.mozilla.org/pub/firefox/releases/55.0.3/source/firefox-55.0.3.source.tar.xz> (Cited from: II)
- [25] LLVM Clang development team. 2018. *LLVM Clang 6.0.0*. Retrieved 2020-01-10 from <https://releases.llvm.org/6.0.0/llvm-6.0.0.src.tar.xz> (Cited from: II)
- [26] MySQL development team. 2018. *MySQL 8.0.11*. Retrieved 2020-01-10 from <https://dev.mysql.com/get/Downloads/MySQL-8.0/mysql-boost-8.0.11.tar.gz> (Cited from: II)
- [27] ScummVM development team. 2018. *ScummVM 2.0.0*. Retrieved 2020-01-10 from <https://www.scummvm.org/frs/scummvm/2.0.0/scummvm-2.0.0.tar.gz> (Cited from: II)
- [28] Todd L. Veldhuizen. 2003. *C++ Templates are Turing Complete*. Technical Report. IndianaUniversity. (Cited from: V-C)
- [29] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. 2003. Issues in Using Students in Empirical Studies in Software Engineering Education. In *Proceedings of the 9th International Symposium on Software Metrics (METRICS 2003)*. IEEE Computer Society, Washington, DC, USA, 239–249. ISBN 0769519873 (Cited from: VI-1)
- [30] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Precise Identification of Problems for Structural Test Generation. In *Proceedings of the 33rd International Conference on Software Engineering (Waikiki, Honolulu, HI, USA) (ICSE 2011)*. ACM, New York, NY, USA, 611–620. ISBN 9781450304450 DOI 10.1145/1985793.1985876 (Cited from: VII)
- [31] Jaroslav Fowkes and Charles Sutton. 2016. Parameter-Free Probabilistic API Mining across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Seattle, WA, USA) (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 254–265. ISBN 9781450342186 DOI 10.1145/2950290.2950319 (Cited from: VII)
- [32] Shay Artzi, Michael Dean Ernst, Adam Kiezun, Carlos Pacheco, and Jeff H. Perkins. 2006. Finding the Needles in the Haystack: GEnerating Legal Test Inputs for Object-Oriented Programs. In *M-TOOS: 1st Workshop on Model-Based Testing and Object-Oriented Systems*. M-TOOS, Portland, OR, USA, 27–34. (Cited from: VII)
- [33] Lajos Cseppentő and Zoltán Micskei. 2017. Evaluating Code-Based Test Input Generator Tools. *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on 27* (2017), 1–24. Issue 6. DOI 10.1002/stvr.1627 (Cited from: VII)
- [34] Andrea Arcuri, Gordon Fraser, and René Just. 2017. Private API Access and Functional Mocking in Automated Unit Test Generation. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Press, Piscataway, NJ, USA, 126–137. DOI 10.1109/ICST.2017.19 (Cited from: VII)
- [35] Hiroaki Yoshida, Guodong Li, Takuki Kamiya, Indradeep Ghosh, Sreeranga P. Rajan, Susumu Tokumoto, Kazuki Munakata, and Tadahiro Uehara. 2017. KLOVER: Automatic Test Generation for C and C++ Programs, Using Symbolic Execution. *IEEE Software* 34, 5 (2017), 30–37. DOI 10.1109/MS.2017.3571576 (Cited from: VII)
- [36] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Computer Aided Verification, Shaz Gopalakrishnan, Ganeshand Qadeer (Ed.)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 609–615. ISBN 9783642221101 (Cited from: VII)
- [37] Hiroaki Yoshida, Susumu Tokumoto, Mukul R. Prasad, Indradeep Ghosh, and Tadahiro Uehara. 2016. FSX: A Tool for Fine-Grained Incremental Unit Test Generation for C/C++ Programs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Seattle, WA, USA) (FSE 2016)*. ACM, New York, NY, USA, 1052–1056. ISBN 9781450342186 DOI 10.1145/2950290.2983937 (Cited from: VII)
- [38] Hiroaki Yoshida, Susumu Tokumoto, Mukul R. Prasad, Indradeep Ghosh, and Tadahiro Uehara. 2016. FSX: Fine-Grained Incremental Unit Test Generation for C/C++ Programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (Saarbrücken, Germany) (ISSTA 2016)*. ACM, New York, NY, USA, 106–117. ISBN 9781450343909 DOI 10.1145/2931037.2931055 (Cited from: VII)
- [39] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2013. Feedback-Directed Unit Test Generation for C/C++ Using Concolic Execution. In *Proceedings of the 2013 International Conference on Software Engineering (San Francisco, CA, USA) (ICSE 2013)*. IEEE Press, Piscataway, NJ, USA, 132–141. ISBN 9781467330763 (Cited from: VII)