Fast Constraint Synthesis for C++ Function Templates

SHUO DING, Georgia Institute of Technology, USA
QIRUN ZHANG, Georgia Institute of Technology, USA

C++ templates are a powerful feature for generic programming and compile-time computations, but C++ compilers often emit overly verbose template error messages. Even short error messages often involve unnecessary and confusing implementation details, which are difficult for developers to read and understand. To address this problem, C++20 introduced *constraints and concepts*, which impose requirements on template parameters. The new features can define clearer interfaces for templates and can improve compiler diagnostics. However, manually specifying template constraints can still be non-trivial, which becomes even more challenging when working with legacy C++ projects or with frequent code changes.

This paper bridges the gap and proposes an automatic approach to synthesizing constraints for C++ function templates. We utilize a lightweight static analysis to analyze the usage patterns within the template body and summarize them into constraints for each type parameter of the template. The analysis is inter-procedural and uses disjunctions of constraints to model function overloading. We have implemented our approach based on the Clang frontend and evaluated it on two C++ libraries chosen separately from two popular library sets: `algorithm` from the Standard Template Library (STL) and `special_functions` from the Boost library, both of which extensively use templates. Our tool can process over 110k lines of C++ code in less than 1.5 seconds and synthesize non-trivial constraints for 30%-40% of the function templates. The constraints synthesized for `algorithm` align well with the standard documentation, and on average, the synthesized constraints can reduce error message lengths by 56.6% for `algorithm` and 63.8% for `special_functions`.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → **Type structures**.

Additional Key Words and Phrases: generic programming, compile-time computations, program analysis

ACM Reference Format:
Shuo Ding and Qirun Zhang. 2025. Fast Constraint Synthesis for C++ Function Templates. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 88 (April 2025), 28 pages. https://doi.org/10.1145/3720422

1 Introduction

C++ is a high-performance programming language widely used in system programming [4], games and GUI [17], compilers [27], artificial intelligence [38], etc. C++ templates are a powerful language feature that facilitates generic programming and compile-time computations, and this feature has been used extensively in practice. It provides compile-time polymorphism, complementing the run-time polymorphism of virtual functions used in many object-oriented languages. For example, almost all containers and algorithms in the Standard Template Library (STL) utilize templates [25]. Template parameters can be values, types, and templates themselves. It is worth noting that templates can be used to simulate arbitrary Turing machines at compile time [57].

During the compilation process, templates are *instantiated* to generate non-templated C++ code: the compiler substitutes formal template parameters with concrete template arguments. If the concrete template arguments do not support certain operations used in the template bodies, the

Authors' Contact Information: Shuo Ding, Georgia Institute of Technology, Atlanta, USA, sding@gatech.edu; Qirun Zhang, Georgia Institute of Technology, Atlanta, USA, qrzhang@gatech.edu.

ACM 2475-1421/2025/4-ART88
https://doi.org/10.1145/3720422

```
#include <vector>

template <typename T> void f(T x) { std::vector<T> v(x, x); }
int main() { f(nullptr); }

/* abbreviated error message (with manually added "...")
old.cc:3:52: error: no matching constructor for initialization of...
template <typename T> void f(T x) { std::vector<T> v(x, x); }
                                                   ^ ~~~~
old.cc:4:14: note: in instantiation of function template special...
int main() { f(nullptr); }
             ^
.../c++/v1/vector:395:57: note: candidate constructor not viable...
(48 more lines of "candidate constructor not viable" or similar) */
```

(a) Erroneous template instantiation without constraints. Apple Clang 15.0.0 with `-std=c++20` prints 55 lines of error messages.

```
#include <vector>
#include <concepts>

template <std::integral T> void f(T x) { std::vector<T> v(x, x); }
int main() { f(nullptr); }

/* abbreviated error message (with some manually added "...")
new.cc:5:14: error: no matching function for call to 'f'
int main() { f(nullptr); }
             ^
...note: candidate template ignored: constraints not satisfied...
template <std::integral T> void f(T x) { std::vector<T> v(x, x); }
(8 more lines) */
```

(b) Erroneous template instantiation with constraints. Apple Clang 15.0.0 with `-std=c++20` prints 13 lines of error messages.

Fig. 1. Erroneous C++ template instantiations without/with constraints.

instantiation generates code that fails to compile, and the compiler emits error messages. Because such failures can occur during deeply nested instantiation processes,[1] it is folklore that C++ template errors could be verbose and difficult to understand [55], and this issue has a long history: for example, for a small 26-byte C++ program, `g++-4.6.3` can produce 15,786 bytes of output, with the longest line of 330 characters [48]. On the other hand, the diagnostics often involve unnecessary implementation details and do not provide much insight into fixing the errors, which confuses C++ developers and hinders production [50, 51]. Consider a simple C++ example in Figure 1a. The `std::vector` class does not have a constructor suitable for the arguments (`nullptr`, `nullptr`). However, C++ compilers are unable to catch this error until the actual instantiation of `std::vector`, resulting in the production of 55 lines of error messages; yet the error messages contain too many implementation details such as "`candidate constructor not viable`" and thus hinder readability.

To improve the readability and maintainability of C++ templates, C++20 introduced a language feature called *constraints and concepts* [43]. Constraints are predicates that impose requirements on template parameters, while concepts are named sets of such requirements. They define clearer interfaces for templates and enable C++ compilers to detect errors early on in the instantiation

[1]The SFINAE mechanism [11] can remove templates from the overload resolution candidate set and thus avoid some errors, but it is often hard to read and maintain.

```
template <typename T>
concept IntClass = requires (T x, T y) {
  typename T::integer_type;
  {x + y} -> std::same_as<T>;
  x.dump();
};

template <typename T> requires IntClass<T> || std::integral<T>
void g(T x) { /* omitted */ }
```

Fig. 2. Complicated constraints with type requirements, compound requirements, simple requirements, and disjunctions of requirement expressions.

process (with better error messages). Consider the example in Figure 1b, which extends the example in Figure 1a by adding the concept `std::integral`. The concept `std::integral`, which is part of the standard library, requires that the template parameters must be of integral types. By using `std::integral`, the error in Figure 1b can be caught before the instantiation of `f`'s body, resulting in only 13 lines of error messages. Furthermore, the message "`constraints not satisfied`" is easily understandable. Note that constraints and concepts are very flexible and expressive. For example, the concept `IntClass` in Figure 2 specifies three requirements on type `T`: (1) it must contain a type member called `integer_type`; (2) it must support the operator `+` with a result type `T`, and (3) it must support the `dump()` member function call. Additionally, `g`'s template parameter `T` must satisfy either the `IntClass` constraint or the `std::integral` constraint. Due to this expressiveness, manually writing and reasoning about constraints and concepts can be error-prone and time-consuming. It becomes more challenging during the development process with frequent code changes. Moreover, many existing C++ projects do not incorporate concepts or constraints, because these language features were only introduced in C++20.

The topic of synthesizing constraints and concepts for C++ templates has not been extensively explored. This paper introduces an approach to automatically synthesize constraints for C++ function templates based on their template bodies and caller-callee relations. Our approach is built on Clang's frontend and leverages only a lightweight static analysis, thus it is very efficient. Moreover, our approach is fully automated and can be directly applied to real C++ code without requiring manual annotations. The synthesized constraints can both specify clearer interface requirements and significantly improve template-related error messages.

Constraint synthesis for C++ function templates is challenging. Because function templates can call other functions or function templates, the synthesis must be inter-procedural to achieve reasonable precision. However, the caller-callee relation can involve arbitrary argument passing and type correspondence. Moreover, C++ supports function overloading, so the actual function being called inside a function template may not be resolved until instantiations. Recursive dependencies, if present, pose yet another challenge. Our approach handles these challenges elegantly and efficiently. We introduce a novel idea called *backmap* to relate type correspondence between the caller and the callee (Section 3.3), use disjunctive clauses to model function overloading (Section 3.3), and cut off recursive dependencies by treating them as trivial constraints (Section 3.4). Finally, we design a polynomial-time simplification procedure for the synthesized constraint formulas (Section 3.5).

At first glance, constraint synthesis for C++ templates bears resemblance to type inference [3, 39, 60]. However, there are several key differences. First, instead of inferring an existing type or typeclass, our approach infers a constraint that corresponds to a set of types, including potential new types that may be defined in the future.[2] Second, the constraints we consider are not limited to those

[2]Section 6.1 discusses a method to match inferred constraints with pre-defined constraints, which can give meaningful names to inferred constraints and help to understand them.

defined by the standard library. In fact, constraints can incorporate arbitrary conjunctive/disjunctive combinations of member function requirements, member type requirements, etc. For example, it is valid to define a constraint requiring the member function `specialFunctionA()`, even if no class in the existing C++ code contains such a member function. Third, as shown in Section 2, the precise requirement of a type can be non-computable due to the potential for a non-terminating C++ compilation if the compiler does not set limits on template instantiations. Therefore, template-related automated reasoning can be viewed as a form of "meta-analysis" of the compilation process.

We have implemented our approach based on the Clang C++ frontend, targeting function templates, supporting constraints in various forms, including unary operators, binary operators, higher-order functions, class member accesses, and simple type traits. We evaluated our tool on real-world library code from the Standard Template Library (STL) header `<algorithm>` and the Boost library[3] header `<boost/math/special_functions.hpp>`. The evaluation results demonstrate that our tool is efficient and effective. Firstly, our analysis is extremely fast, taking less than 1.5 seconds to process over 110k lines of code (LOC). We are able to synthesize non-trivial constraints for 38.4% of function templates from `algorithm` and for 35.9% of function templates from `special_functions`. Secondly, our analysis is reasonably precise. We select the 14 representative function templates from `algorithm`, as listed in the introductory C++ textbook [53]. We compare the synthesized constraints with the standard requirements specified in the document. The majority of the synthesized constraints either match or under-approximate the standard requirements. Finally, the synthesized constraints significantly reduce the length of compiler error messages for incorrect instantiations of function templates, with average reductions of 56.6% for `algorithm` and 63.8% for and `special_functions`.

This paper makes the following contributions.

- We study and analyze the problem of synthesizing template constraints for improving C++ code readability and maintainability.
- We present an automated constraint synthesis for C++ templates.
- We conduct an extensive evaluation based on two widely used C++ libraries. The empirical results demonstrate that our approach is fast, precise, and can significantly reduce template-based compiler errors.
- We designed and implemented a way to automatically measure the effectiveness of compilation error message reductions.

The rest of the paper is structured as follows. Section 2 describes the C++ background and the problem that we target, including its general undecidability. Section 3 describes our approach in detail. Section 4 gives the experimental results. Section 5 contains three case studies for error messages. Section 6 discusses more about the spirit of our work and technical details. Section 7 surveys related work, and Section 8 concludes.

2 Preliminary

This section reviews the background and introduces the constraint synthesis problem.

2.1 C++ Function Templates, Constraints, and Concepts

In C++, a *function template* F defines a family of functions. Abstractly, F takes a list of template arguments $\vec{a}$ and returns a concrete C++ function $F(\vec{a})$; this computation, normally called template instantiation, is done in compile-time. Since template arguments $\vec{a}$ might result in type errors after the instantiation of function template F, a *constraint* can be associated with F to specify requirements on F's template arguments. A named set of such constraints is called a *concept*. A

[3] https://www.boost.org.

```
template <typename T>
void f(T x) {
    for (int i = 0; i < 3; i++)
        x.dump();
}

template <typename U, typename V>
void g(U x) {
    x.print(100);
    for (int i = 0; i < 10; i++)
        x.print(i);
    V y;
    f(y);
}
```

(a) An example of unconstrained C++ function templates.

```
template <typename T>
concept Dumpable =
    requires (T x) { x.dump(); }

template <Dumpable T>
void f(T x) {
    for (int i = 0; i < 3; i++)
        x.dump();
}

template <typename U, typename V>
requires Dumpable<V> &&
  (requires (U x, int y) { x.print(y); })
void g(U x) {
    x.print(100);
    for (int i = 0; i < 10; i++)
        x.print(i);
    V y;
    f(y);
}
```

(b) The syntax of constraints and concepts in C++20.

Fig. 3. The syntax of C++ templates, constraints, and concepts.

concept C is a compile-time predicate taking a list of template arguments $\vec{a}$ and returning `true` or `false`, and C can be used to specify requirements for multiple function templates.

We use an example to illustrate C++ templates and constraints/concepts. The most common syntax of C++ function template definition consists of a template parameter list followed by the function body, as shown in Figure 3a. In this example, the template parameters `T`, `U`, and `V` are unconstrained. Specifically, the keyword **`typename`** only indicates that these template parameters should be "types." However, it is straightforward to see that not all types can be used:

- a variable of type `T` must support the member function call `dump()`;
- a variable of type `U` must support the member function call `print(int)`;
- type `V` must at least satisfy the constraint of `T`.

To express these constraints, we can either define a standalone concept (`Dumpable`) and replace **`typename`** with it, or use the **`requires`** clause to specify the constraints in-place, as shown in Figure 3b. Here the concept `Dumpable` is a named predicate checking whether a variable of the given type supports the `dump()` member function call, and **`requires`** is used both to associate constraints (either pre-defined concepts or directly written constraint expressions) to function templates and to start "requires expressions" that can be used as parts of larger constraints.

2.2 Problem Statement and Undecidability

From Figure 3, we can see that the process of writing constraints involves inter-procedural reasoning. For example, in Figure 3b, the requirement `Dumpable` originates from `f` and is propagated into `g`, because `g` calls `f`. Real-world C++ code consists of much more complicated function templates and call graphs (with possibly overloaded callees), and during the development process, frequent code changes make the situation harder. Thus, manually completing the above process is non-trivial and time-consuming. This paper proposes automated C++ template constraint synthesis to help the development process. We focus on type parameters of templates because that is the most frequently used feature in generic programming like STL.

Ideally, our goal is to precisely define the set of requirements for each template type parameter. However, because C++ template is Turing-complete, we can demonstrate that precise constraint

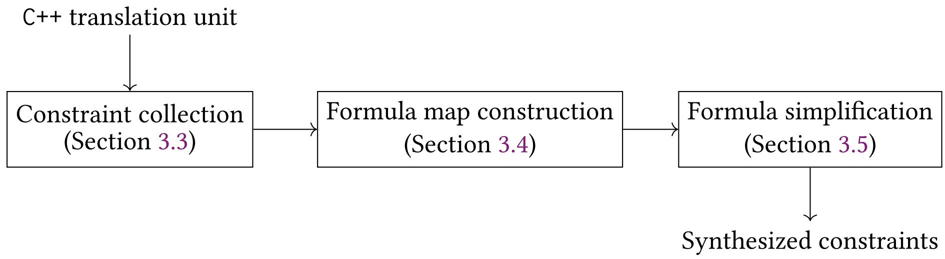


Fig. 4. An overview of our approach. First, constraint collection (Section 3.3) traverses each function template to collect constraints into the inter-procedural constraint map. Second, formula map construction (Section 3.4) takes the inter-procedural constraint map and produces the formula graph, which is a compact representation of all constraints and their dependencies in the entire translation unit. Finally, formula simplification (Section 3.5) uses a lightweight algorithm to simplify the constraints into versions that are suitable to be inserted into the source code.

synthesis is undecidable by slightly modifying the construction proposed by Veldhuizen [57]. Specifically, given an arbitrary closed Turing machine M, we define a `struct S` containing a member type `halted_state`, so that in a specific instantiation of `S`, the `halted_state` is

- `int` if and only if M halts in the accept state,
- `double` if and only if M halts in the reject state,
- undefined if M does not halt at all.

It is well known that there does not exist a total program that distinguishes the first two cases even if we allow the program to output arbitrary results for the third case [20]. We can further specify constraints on any type template parameter

```
static_assert(std::is_same<S<...>::halted_state, T>::value);
```

so in general precise constraint synthesis for type template parameters is also non-computable, and practical synthesizers must sacrifice precision in order to guarantee termination. Our work synthesizes *under-approximations* of the precise requirements. An under-approximation of requirements means specifying fewer requirements, which corresponds to allowing more types. To sum up, we give our problem definition as follows. In particular, we mainly target `C++` programs that do not contain any constraints, so there is no interference between existing constraints and our constraint generation process.

> Given a `C++` translation unit, for each function template in the translation unit, for each type template parameter of the function template, compute a constraint C under-approximating the real requirement $\mathcal{R}$ on this parameter. C is specified using the `C++20` constraints and concepts syntax, and $\mathcal{R}$ is the requirements on the parameter implicit in the template body. Suppose the set of types allowed by C and $\mathcal{R}$ are $S(C)$ and $S(\mathcal{R})$, respectively. An under-approximating constraint allows more types, so we can express our goal as $S(C) \supseteq S(\mathcal{R})$.

3 Approach

This section formalizes our approach in detail. To aid the formal discussion, Section 3.1 introduces a simplified calculus for modelling `C++` function templates; Section 3.2 introduces *constraint formulas* which are finally inserted into the source code to restrict type template parameters. Our main approach consists of the following three steps, which is summarized in Figure 4.

$$
\begin{array}{rcl}
T & \in & \text{TypeParameter} \\
t & \in & \text{Type} \cup \text{TypeParameter} \\
v & \in & \text{Variable} \\
n & \in & \text{FunctionOrFieldName} \\
e & \in & \text{Expression} \\
\text{op}_\text{p} & \in & \text{PrefixOperator} \\
\text{op}_\text{s} & \in & \text{SuffixOperator} \\
\text{op}_\text{i} & \in & \text{InfixOperator} \\
\text{trait} & \in & \text{TypeTrait} \\
\text{use} & := & \text{op}_\text{p}\ v \mid v\ \text{op}_\text{s} \mid v\ \text{op}_\text{i}\ e \mid e\ \text{op}_\text{i}\ v \mid v(e^*) \mid v.n \mid v.n(e^*) \mid \text{trait}(t) \mid n(\ldots v \ldots) \\
\text{fun} & := & t\ n((t\ v)^*)\ \{\text{use}^*\} \\
\text{temp} & := & <T^+>\ \text{fun} \\
\text{spec} & := & <>\ \text{fun} \\
\text{translationUnit} & := & (\text{fun} \mid \text{temp} \mid \text{spec})^+
\end{array}
$$

Fig. 5. A simplified calculus for modelling C++ function templates.

(1) Section 3.3 describes the process of scanning template bodies and constructing the *inter-procedural constraint map* for each translation unit.
(2) Section 3.4 transforms the inter-procedural constraint map to the *formula graph*, during which recursive dependencies are resolved.
(3) Section 3.5 describes the simplification process after directly reading the constraint formulas from the formula graph.

Section 3.6 discusses the soundness aspect of our approach.

3.1 A Simplified Calculus

C++ is notorious for its complex syntax and semantics, as evidenced by the 1853-page C++20 standard [23]. To enable a formal discussion, we present a simplified calculus that models the core semantics of C++ function templates. This is similar to the spirit of Garcia and Lumsdaine [19], but we also choose to adhere more to the realistic syntax of C++ since our approach applies to real C++ code. On syntax constructs of the simplified calculus that deviate from the standard C++ language, our open-source implementation provides the complete details.

Figure 5 gives the syntax of our simplified calculus. As usual, "$*$" means repeating zero or more times and "+" means repeating one or more times. The ellipsis "$\ldots$" means similar but irrelevant syntax constructs, such as the argument list $(\ldots v \ldots)$ where we only consider the argument v and ignore other arguments. We consider type template parameters "T", and "t" represent concrete types or type template parameters. Variables "v" and expressions "e" represent standard C++ variables and expressions, respectively. We use "n" to denote function or field names. We consider three types of operators because these are the most common ones in C++: prefix operators (op_p), suffix operators (op_s), and infix operators (op_i). C++ type traits (trait) are template-based utilities that can provide type information at compile-time.

Our synthesis framework is flow-insensitive, so we omit control flows and only consider the usage sites "use" of type template parameter t or variable v of type t. The usage sites include direct trait assertions on the type template parameter ($\text{trait}(t)$), and usages of variables of the type template parameter, such as being used as operands ($\text{op}_\text{p} v$), functions ($v(e^*)$), member accesses ($v.n$, $v.n(e^*)$), and function call arguments ($n(\ldots v \ldots)$, where v should satisfy n's corresponding type requirements). A function body "fun" could be non-template functions where the types t involved are all concrete types or template functions where the types t could be type template parameters. A function template "temp" consists of a list of type template parameters and a function body. A function template can have specializations "spec" where all type template parameters are substituted by concrete types. All of "fun", "temp", and "spec" participate in overloading resolution of function calls. Finally, a "translationUnit" is a basic compilation unit for this calculus. Our analysis is performed on individual translation units. Our implementation is based on Clang, and Clang has all the classes corresponding to real `C++` constructs, such as `TemplateTypeParmDecl`, `DeclRefExpr`, and `TemplateSpecializationType`, etc.

3.2 Constraint Formalization

`C++20` supports many kinds of constraints, such as type constraints (*e.g.* requiring the existence of a type member), expression constraints (requiring an expression such as `a.f(1, true)` to successfully compile), type traits (*e.g.* `std::is_same`). These are considered as *atomic constraints*. Our work also supports conjunctions and disjunctions of smaller constraints according to `C++20`. Formally, we define *constraint formulas* as follows.

Definition 3.1 (Constraint formula). Constraint formulas express constraints on types. Atomic formulas pose restrictions on the type template parameter T, and compound formulas are either atomic formulas or conjunctions/disjunctions of smaller formulas. The atomic formulas are similar to the use part of our simplified calculus in Figure 5, except that we only preserve the type information (e.g., the exact expression e in v op$_i$ e is omitted; only its type is preserved). Note that a specific atomic formula convertible_to is introduced, which helps to handle overloaded callee candidates where implicit type conversions are permitted.

$$\begin{array}{rcl} \text{atomic} & := & \mathrm{op_p}\ T \mid T\ \mathrm{op_s} \mid T\ \mathrm{op_i}\ t \mid t\ \mathrm{op_i}\ T \mid T(t^*) \mid T.n \mid T.n(t^*) \mid \\ & & \mathrm{trait}(T) \mid \mathrm{convertible\_to}(T, t) \\ \text{formula} & := & \text{atomic} \mid (\wedge\ \text{formula}^*) \mid (\vee\ \text{formula}^*) \end{array}$$

Example 3.2. In real `C++` code, the constraint formula `(T x) { ++x; } && ((T x) { x.f(); } || (T x) { x.g(); })` consists of three atomic constraints `(T x) { ++x; }`, `(T x) { x.f(); }`, and `(T x) { x.g(); }`. The overall requirement is that the type `T` must support the prefix-increment operator `++`, and must support member function calls of either `f()` or `g()`. Note that `(T x) { ++x; }` corresponds to the atomic constraint $++T$ as defined in Definition 3.1, and thus Definition 3.1 can be regarded as abbreviations of real `C++` constraints.

We do not reason about implications between different atomic constraints except for equality comparisons. Certain atomic constraints are *normalized* to reduce redundancies: for example, `(T x, T y) { x + y; }` and `(T x, T y) { y + x; }` are treated as the same atomic constraint by rewriting $y + x$ to $x + y$ (i.e. the x is always on the left hand side). Our main reasoning effort is devoted to conjunctions and disjunctions, *e.g.* removing duplicate conjuncts. Also, we consider constraint formulas for each template type parameter T, but the formula itself can involve other type parameters, such as `std::same_as<T, U>`, which is an atomic constraint formula requiring T to be the same as another template type parameter U.

3.3 Interprocedural Constraint Map Construction

The first step is to traverse the abstract syntax tree (AST) of the `C++` translation unit and construct the *inter-procedural constraint map*[4] representing both intra-procedural constraints inside individual

[4]The same idea can be extended to class templates, which could be defined as *inter-class constraint map*.

$$\frac{\mathsf{op_p}\ v \quad v : T}{\mathsf{op_p}\ T}\,(1) \qquad \frac{v\ \mathsf{op_s} \quad v : T}{T\ \mathsf{op_s}}\,(2) \qquad \frac{v\ \mathsf{op_i}\ e \quad v : T \quad e : t}{T\ \mathsf{op_i}\ t}\,(3) \qquad \frac{e\ \mathsf{op_i}\ v \quad e : t \quad v : T}{t\ \mathsf{op_i}\ T}\,(4)$$

$$\frac{v(e^*) \quad v : T \quad e^* : t^*}{T(t^*)}\,(5) \qquad \frac{v.n \quad v : T}{T.n}\,(6) \qquad \frac{v.n(e^*) \quad v : T \quad e^* : t^*}{T.n(t^*)}\,(7) \qquad \frac{\mathsf{trait}(T)}{\mathsf{trait}(T)}\,(8)$$

$$\frac{n(\ldots v \ldots) \quad v : T \quad (< \ldots U \ldots > \_\, n(\ldots U \_ \ldots)\ \{\ldots\})^* \quad ((<>)?\ \_\, n(\ldots t \_ \ldots)\ \{\ldots\})^*}{[(U, m)^*, t^*]}\,(9)$$

Fig. 6. Rules for inter-procedural constraint map construction.

function templates and the relations between different function templates. The traversal can be either depth-first search or breadth-first search, and we primarily collect all usage sites of variables of types in the template parameter list. Formally, for each C++ translation unit $\mathcal{U}$, we use $\mathcal{U}_F$ to denote the set of function templates in $\mathcal{U}$. For each function template $f_i \in \mathcal{U}_F$, we use $\mathbf{TTParm}(f_i)$ to denote the set of type template parameters of f_i. Given a translation unit $\mathcal{U}$, we define the *inter-procedural constraint map* $M_{\mathcal{U}}$, which maps each type template parameter in $\bigcup_{f_i \in \mathcal{U}_F} \mathbf{TTParm}(f_i)$ to a set of constraints that it should satisfy.

Definition 3.3 (Inter-procedural constraint map). Given a specific type template parameter $T \in \mathbf{TTParm}(f_i)$, we classify the constraints in $M_{\mathcal{U}}(T)$ into two categories.

- **Intraprocedural Constraints:** This category includes constraints on T that are not dependent on other functions or function templates, such as unary/binary operators, member accesses, etc., inside the body of f_i. They are atomic constraints as defined in Definition 3.1.
- **Interprocedural Constraints:** An inter-procedural constraint is generated for each named call-site $g(\ldots)$ inside f_i where a variable of type T is used as the k-th argument. Because of overloading, g can refer to many functions or function templates sharing the same name: $\{g^0, g^1, \ldots\}$. The inter-procedural constraint for this call-site is thus a list of elements in the following forms.
  - (U, m) corresponds to another function template g^j, such that the k-th parameter of g^j is of type $U \in \mathbf{TTParm}(g^j)$. m is a corresponding *backmap* which will be explained later. Overall, this means T should satisfy whatever U satisfies.
  - t corresponds to a function or function template specialization[5] g^k, such that the k-th parameter of g^k is of concrete type t; or a function template g^k but the k-th parameter of g^k is of concrete type t. Overall, this means T should be convertible to t.

Constraint Collection. Figure 6 depicts our constraint collection process using typing rules. Note that, unlike conventional typing rules that start with type environments, our implementation directly uses the Clang frontend to obtain types for variables and expressions. In particular, Clang's `Expr` class[6] has a member function `getType`, which returns the (possibly qualified) type `QualType` of the expression. As a result, we omit type environments and focus on constraint generations. In terms of formalization, we can consider an elaborated version of our simplified calculus with explicit type annotations for variables and expressions, as shown in Figure 7.

[5] C++ only allows full specialization for function templates, meaning that every template parameter should be concrete in the specialization.
[6] https://clang.llvm.org/doxygen/classclang_1_1Expr.html.

$$
\begin{array}{rcl}
T & \in & \text{TypeParameter} \\
t & \in & \text{Type} \cup \text{TypeParameter} \\
v : t & \in & \text{Variable} \\
e : t & \in & \text{Expression}
\end{array}
$$

Fig. 7. The elaboration of our simplified calculus where types of variables and expressions are explicitly annotated. We only show the most relevant parts (*i.e.* variables and expressions) of the language definition, as other parts follow Figure 5 with v and e replaced by type-annotated versions.

The formal constraint collection process is shown in Figure 6, where T is the type template parameter for which we want to infer constraints. $*$ means repeating zero or more times, "_" means "don't care" tokens (i.e., source code tokens that are ignored), and "..." means adjacent irrelevant syntax constructs.

- Rules (1) - (4) consider the variable v of type T used in expressions. For example, if v is used as "++v," we generate an atomic constraint "++T."
- Rule (5) considers the variable v of type T used as a higher-order call, meaning that T must be a callable type.
- Rules (6) - (7) consider the variable v of type T where a member access is requested, meaning that T should have the corresponding members.
- Rule (8) considers type trait predicates directly applied to the type template parameter T.
- Rule (9) considers the variable v of type T used as an argument for a function call, where v must satisfy the corresponding requirements of at least one overloading candidate.

Note that Figure 6 only shows the rules for our simplified calculus for demonstration purposes, and in our implementation for the C++ language, we have further ad-hoc reasoning processes to refine the results. For example, if a higher-order function call (Rule (5) in Figure 6) is used as a condition of a branch, then its return type should be convertible to the Boolean type, which corresponds to an additional constraint.

The constraints collected in this stage are not completely in the form of constraint formulas in Definition 3.1, because the inter-procedural constraints (Rule (9) in Figure 6) are not defined in Definition 3.1 and we have not generated conjunctions or disjunctions. In the formula map construction stage in Section 3.4, (unsimplified) constraint formulas as defined in Definition 3.1 are generated, where conjunctions are constructed to model multiple requirements in the same function template, and disjunctions are constructed to model function overloading (Rule (9) in Figure 6).

Constraint Propagation. To correctly propagate constraints inter-procedurally, we need *backmaps*. We explain the intuition through an example. Consider the following C++ code.

```
template <typename T> void f(T x) { x++; }
template <typename U> void g(U x) { f(x); }
```

We need to propagate the constraint on the template parameter T of f, which is `(T x) { x++; }`, to the template parameter U of g. However, we cannot directly copy that constraint, because at g, there is no type template parameter named T. To resolve this issue, we design backmaps, which store what arguments (in this case, U) are substituted for the callee's type template parameter (in this case, T). In a more general case where we have a chain of function calls (f_1 calls f_2, f_2 calls f_3, etc.) of length l, the correct type can be resolved by iteratively stepping through the l backmaps.

Definition 3.4 (Backmap). For each named call-site $g(\ldots)$ inside a function template f, for each function template g^i in the overloading candidate set $\{g^0, g^1, \ldots\}$, the *backmap* m_i is defined as a (possibly non-total) map mapping each type template parameter $U \in \mathbf{TTParm}(g^i)$ to either a concrete type t or a type depending on template parameters in $\mathbf{TTParm}(f)$.

In our implementation, we construct backmaps in a best-effort fashion. This involves analyzing the named call-site and identifying the arguments that should be used to replace the type template parameters of the callee. If the correct type cannot be resolved, then we simply discard the constraint, which still preserves the under-approximation property.

Example 3.5 (Inter-procedural constraint map and backmap). For the following code,

```
void f(int x) {}
template <typename T> void f(T x) { x++; ++x; x+=1; }
template <typename U> void g(U x) { f(x); x.print(); }
```

the corresponding inter-procedural constraint map is

$$\left\{ \begin{array}{ll} T & \rightarrow \{ \texttt{(T x)\{x++;\}},\ \texttt{(T x)\{++x;\}},\ \texttt{(T x,int y)\{x+=y;\}} \} \\ U & \rightarrow \{ [\texttt{int}, (T, m)],\ \texttt{(U x)\{x.print();\}} \} \end{array} \right\}$$

where $[\texttt{int}, (T, m)]$ is an inter-procedural constraint as described in Definition 3.3, and the corresponding backmap m for the function template `f` (not the separate overloading of `f` with concrete type `int`) is

$$\{T \rightarrow U\}.$$

Note that in our implementation, the keys of constraint maps are pointers to template type parameters in the AST (`const clang::TemplateTypeParmDecl*`), so there is no ambiguity even if different function templates use the same name (such as T) for their template parameters.

3.4 Formula Map Construction

The inter-procedural constraint map M contains all we need for constraint synthesis, but recursive dependencies could exist. For example, the constraints of a function template's type parameter T can depend on another function template's type parameter U, which can, recursively, depend on T again. Our second step is thus obtaining the *formula map*, which maps type template parameters to constraints formulas, and which does not contain recursive dependencies. Algorithm 1 gives the formula map construction algorithm, which employs a depth-first search on the inter-procedural constraint map.

Algorithm 1 maintains two data structures. The *status* map at line 3 represents whether the constraint formula of a type template parameter has not been touched by the algorithm (NOTVISITED), is in the process of being constructed (ONSTACK), or has already been constructed (VISITED). The *result* map at line 5 represents the formula map being computed, wherein the implementation, we actually store the pointers to constraint formulas, so that the same constraint formula for a type template parameter of a callee can be shared by different callers. The function *DFS* at line 6 recursively construct the constraint formula for each type template parameter (line 31), and any recursive dependency in the inter-procedural constraint map is truncated at line 27 and is eventually treated as `true` (line 18). Inside *DFS*, the only case where we involve backmaps is for inter-procedural constraints of the form (U, m), where we also store the backmap to the constraint formula being constructed at line 20 so that we can recover type names for inter-procedural constraints.

The order of visiting keys (line 31) can affect the final results. Consider this chained dependency of type template parameters $T \leftarrow U \leftarrow V \leftarrow T$, meaning that T depends on U, U depends on V, and V depends on T again. If T is visited first, then when we recursively get to V, V's dependency of T will be truncated to true; if V is visited first, then V's dependency of T will at least include the intra-procedural constraints of T. This is a loss of precision when handling recursions, and we

Algorithm 1 The formula map construction algorithm

```
function constructFormulaMap(M /* inter-procedural constraint map */)
    // type template parameter → construction status
    status ← emptyMap(default = NOTVISITED)
    // type template parameter → formula
    result ← emptyMap()
    function dfs(T /* type template parameter */)
        if status[T] = NOTVISITED then
            status[T] ← ONSTACK
            conjunction ← emptyConjunction()
            for constraint ∈ M[T] do
                if constraint.intra() then // intra-procedural constraint
                    conjunction.add(constraint)
                else if constraint.inter() then // inter-procedural constraint
                    disjunction ← emptyDisjunction()
                    for (U, m) ∈ constraint do
                        temp ← dfs(U)
                        if temp = NULL then
                            disjunction.add(true)
                        else
                            disjunction.add((m, temp))
                    for t ∈ constraint do
                        disjunction.add(convertible_to(T, t))
                    conjunction.add(disjunction)
            result[T] ← conjunction
            status[T] ← VISITED
            return conjunction
        else if status[T] = ONSTACK then
            return NULL
        else if status[T] = VISITED then
            return result[T]
    for T ∈ M.keys() do
        if status(T) = NOTVISITED then
            dfs(T)
    return result
```

choose to keep the algorithm lightweight without introducing time-consuming processes such as fixed point computations.

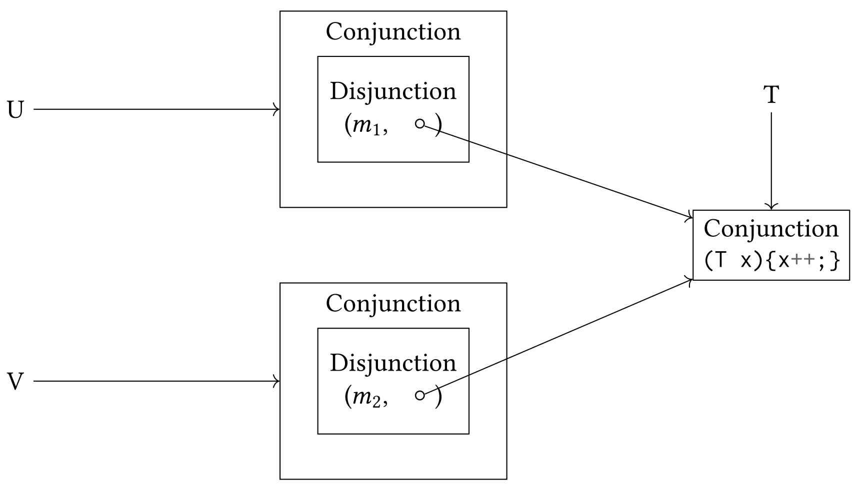
Example 3.6 (Formula map). For the C++ code shown in Figure 8a, the corresponding formula map is shown in Figure 8b, where m_1 and m_2 are backmaps for call-site 1 and call-site 2, respectively. The constraint of T is also shared inside U and V's constraints, because g1 and g2 both call f. Note that there are redundant nested conjunctions and disjunctions in this case, which can be handled by our formula simplification algorithm in Section 3.5.

3.5 Formula Simplification

From the formula map, we can recursively read and print the constraint formula (as defined in Definition 3.1) for each type template parameter, with the help of backmaps. However, because of chained function calls in function templates, the constraint formulas could contain redundancies

```
template <typename T> void f(T x) { x++; }
template <typename U> void g1(U x) { f(x); // call-site 1 }
template <typename V> void g2(V x) { f(x); // call-site 2 }
```

(a) Three C++ function templates where g1 and g2 share the callee f.



(b) The corresponding formula map where T, U, V are type template parameters pointing to their constraints.

Fig. 8. A piece of C++ code and its corresponding formula map.

such as $(\wedge(\vee(\wedge(\vee(\text{atomicConstraint})))))$. A typical case is that when there is only one callee for a call-site inside the template body, Algorithm 1 still insert a disjunction layer in the formula. To make the final constraint formula smaller, we apply a simple polynomial-time simplification process (Algorithm 2) before actually inserting the constraint formula into C++ code.

The basic idea of Algorithm 2 is to simplify conjunctions (line 1) and disjunctions (line 27) in a bottom-up fashion using recursion. Redundant layers of the formula are recursively eliminated (line 7). The main simplification rules include de-duplicating terms and discarding trivial terms (line 12). The main procedure (line 29) starts the simplification based on the input formula's format (atomic, conjunction, disjunction).

There exist sophisticated Boolean formula minimization algorithms, such as the Quine-McCluskey algorithm [35, 40, 41], but the problem itself is NP-hard [56]. Since our goal is to keep our approach lightweight, we choose to use our Algorithm 2. Suppose the input formula f's length is l and nesting depth is d, and suppose the hash function takes linear time with respect to the input length. Then Algorithm 2's time complexity is $O(ld)$, and is thus polynomial time with respect to the formula size.

3.6 Soundness versus Soundiness

Our goal is to let the computed constraint formula C under-approximate (allow more types than) the real requirement $\mathcal{R}$. Suppose the set of types (including new types that can be added to the code in the future) allowed by C and $\mathcal{R}$ are denoted as $S(C)$ and $S(\mathcal{R})$, respectively; as we discussed in Section 2, we need $S(C) \supseteq S(\mathcal{R})$. We call this property *soundness*. In our simplified calculus (Figure 5), soundness is ensured by our algorithms.

Algorithm 2 The formula simplification algorithm

```
function simplifyConjunction(f /* constraint formula */)
    newConjunction ← emptyConjunction()
    deduplicate ← emptyHashSet()
    for conjunct ∈ f do
        candidates ← emptyList()
        conjunct ← simplifyFormula(conjunct) // recursive simplification
        if conjunct.isConjunction() then // inspect one layer of conjunction
            for child ∈ conjunct do
                candidates.add(child)
        else
            candidates.add(conjunct)
        for candidate ∈ candidates do
            if candidate.isTrue() then // omit trivial conjuncts
                continue
            else if candidate.isFalse() then // f is trivially false
                return false
            else
                if not deduplicate.contains(candidate.hash()) then
                    newConjunction.add(candidate)
                    deduplicate.add(candidate.hash())
    if newConjunction.length() = 0 then // f is trivially true
        return true
    else if newConjunction.length() = 1 then // omit one trivial conjunction layer
        return newCojunction.first()
    else
        return newCojunction
function simplifyDisjunction(f /* constraint formula */)
    // similar to simplifyConjunction, omitted
function simplifyFormula(f /* constraint formula */)
    if f.isAtomic() then
        return f
    else if f.isConjunction() then
        return simplifyConjunction(f)
    else// disjunction
        return simplifyDisjunction(f)
```

Theorem 3.7. *For programs written in the simplified calculus shown in Figure 5, for every type template parameter T, if a type argument t is passed in for T and does not result in compile-time errors, then the constraint formula generated for T according to the three steps (Sections 3.3, 3.4, and 3.5) evaluates to true on t.*

Proof. Since the type argument t does not result in compile-time errors, all uses of type t in the code are valid. Every intra-procedural constraint as defined in Definition 3.3 is satisfied, so the conjunction of these constraints is also satisfied. Every inter-procedural constraint, as defined in Definition 3.3, according to overloading resolution, should result in at least one valid candidate, so the disjunction of the constraint formulas from the overloading candidates is satisfied. The choice of truncating recursive dependencies to `true` in Section 3.4 only relaxes the constraint, so

```
#include <utility>

struct S { void f() && {} };

template <typename T>
void g(T x) { std::move(x).f(); }

int main() { g(S{}); }
```

(a) Original code: successful compilation on Apple clang 15.0.0.

```
#include <utility>

struct S { void f() && {} };

template <typename T>
requires requires (T o) { o.f(); }
void g(T x) { std::move(x).f(); }

int main() { g(S{}); }
```

(b) Modified code: failed compilation on Apple clang 15.0.0.

Fig. 9. An unsound corner case where our tool inserted over-constrained constraints. The member function `f` of `S` should be invoked on r-values, while our tool ignores references and uses an l-value to invoke `f` in the constraint. Note that **`requires requires`** is not a typo: the first **`requires`** specifies the constraint for the template while the second **`requires`** starts a constraint expression.

satisfaction is preserved. The formula simplification algorithm as described in Section 3.5 preserves the truth value of the constraint formulas, so the simplified formula is also satisfied by t. ◻

However, almost all realistic static analysis tools are unsound in certain aspects [32]. The reasons include scalability, precision, and engineering details of realistic programming languages. A static analysis tool is *soundy* when most common language features are soundly-approximated while some special language features, well-known to experts in the area, are unsoundly-approximated [32]. This is known as the *soundiness*. We claim our implementation is soundy for C++. First, we apply the following general strategy in our implementation: whenever we encounter unsupported C++ language features, we treat the constraint generated by the unsupported part as `true`, so our tool gracefully bypasses them and approximates toward $S(C) \supseteq S(\mathcal{R})$.

- For example, `std::enable_if` expressions occurring on parameter lists and/or return types are currently ignored, which relaxes the constraints generated by our tool and preserves under-approximation. We plan to support `std::enable_if` by starting from simple cases (such as single traits `std::is_void<T>::value` or its negation) and gradually handling more and more complicated constraints (including nested expressions).
- As another example, if one of the overloading candidates contains unsupported features such as **`if constexpr`**, then the entire constraint for the candidate falls back to `true`, which still preserves under-approximation.

Second, for cv(const/volatile)-qualifiers and lvalue/rvalue references, with deliberately designed corner cases such as Figure 9, our tool can generate slightly over-constrained constraints and therefore results in $S(C) \subset S(\mathcal{R})$. In Figure 9, the reason is that we ignore references when handling constraints. To sum up, our tool is sound except for the two features (1) cv-qualifiers and (2) lvalue/rvalue references. Those two features have complicated interactions with template argument deductions [8, 12], and we leave the completely sound handling of them as future work.

4 Experiments

We implemented our approach based on the `RecursiveASTVisitor` facility from Clang frontend (forked from the main branch of Clang in October 2023). Our implementation language is C++, consisting of roughly 2.8 kLOC. We support unary operators, binary operators, higher-order functions, class member accesses, and simple type traits as atomic constraints. We provide the link to our open-source implementation at the end of this paper.

Table 1. Overall performance. The execution time includes not only the three steps described in Section 3, but also the actions of generating the rewritten code, reporting statistical results, etc.

	algorithm	special_functions
Total LOC after preprocessing	32,206	111,862
Execution time (seconds)	0.250	1.302
Number of templates	821	2,531
Number of templates with nontrivial results	315	908

```
template <typename _Integral>
__attribute__((__visibility__("hidden")))
__attribute__((__exclude_from_explicit_instantiation__))
__attribute__((__abi_tag__("v160006"))) constexpr
typename enable_if
<
    is_integral<_Integral>::value,
    _Integral
>::type
__half_positive(_Integral __value)
{
    return static_cast<_Integral>(
        static_cast<__make_unsigned_t<_Integral> >(__value) / 2
    );
}
```

Fig. 10. A function template from `algorithm` that our tool fails to synthesize non-trivial constraints.

We conducted experiments on two libraries: `<algorithm>` from the Standard Template Library (STL) and `<boost/math/special_functions.hpp>` from the Boost library. We chose these two libraries because they mainly consist of function templates instead of class templates, and our tool currently only targets function templates, although the idea can also be extended to class templates. Our evaluation focuses on three dimensions.

- *Performance (Section 4.1).* For both libraries, we report the numbers of function templates for which our tool can infer nontrivial constraints (constraints that are not simply the literal `true`). We also measure execution time.
- *Precision (Section 4.2).* For the `algorithm` library, we compare the inferred constraints with the documented constraints.
- *Error Reduction (Section 4.3).* For both libraries, we measure the Clang compilation error message reductions after adding constraints.

It is important to mention that our tool is lightweight and can be used on ordinary hardware. To demonstrate this, all of our experiments were conducted on a MacBook Air (2020) with Apple M1 chip and 8GB memory, and everything was executed in a single thread. The pre-processing of standard library headers and the compilation error measurements were all based on the main compiler on the MacBook, which is Apple Clang version 15.0.0.

4.1 Overall Performance

Table 1 presents the execution speed and the number of function templates with non-trivial synthesized constraints. Our analysis is highly efficient, taking only 0.250 seconds to process more than 30k lines of code from `algorithm` and only 1.302 seconds to process over 110k lines of code from `special_functions`. Note that these times include not only the three steps described in Section 3, but also the reporting of statistical results, and the code rewriting for adding synthesized constraints. This demonstrates the exceptional performance of our tool. Furthermore, our tool can

Table 2. Precision on STL `algorithm` library. The library requirement is obtained from the standard document, while the synthesized requirement is generated by our tool.

Template	Documented requirement	Synthesized requirement
`for_each`	(InputIterator, UnaryFunction)	(Iterator, UnaryFunction)
`find`	(InputIterator, Any)	(Iterator, Any)
`find_if`	(InputIterator, Predicate)	(Iterator, Predicate)
`count`	(InputIterator, Any)	(Iterator, Any)
`count_if`	(InputIterator, Predicate)	(Iterator, Predicate)
`replace`	(ForwardIterator, Any)	(Iterator, Any)
`replace_if`	(ForwardIterator, Predicate, Any)	(Iterator, Predicate, Any)
`copy`	(InputIterator, OutputIterator)	(Any, Any)
`copy_if`	(InputIterator, OutputIterator, Predicate)	(Iterator, Iterator, Predicate)
`move`	(InputIterator, OutputIterator)	(Any, Any)
`unique_copy`	(InputIterator, OutputIterator)	(Iterator, Iterator)
`sort`	(ValueSwappable ∧ RandomAccessIterator)	(RandomAccessIterator)
`equal_range`	(ForwardIterator, Any)	(Iterator, Any)
`merge`	(InputIterator, InputIterator, OutputIterator)	(Iterator, Iterator, Iterator)

synthesize non-trivial constraints for approximately 30%-40% of function templates. The remaining function templates either (1) lack requirements in our supported categories of atomic constraints, or (2) incorporate C++ features that are not yet supported by our tool, such as complicated type sugars or aliases. Note that the two cases are overlapping (there could be function templates incorporating unsupported features while at the same time does not have constraints in reality), and in general it is not possible to automatically isolate the first case due to the undecidability mentioned in Section 2.2. An example from the `algorithm` library in which our tool fails to synthesize constraints is shown in Figure 10, where the usage of the variable `__value` is wrapped in type conversions before being divided by the operator `/`, and our tool currently only supports limited forms of such wrappers. It is worth mentioning that our conservative strategy (Section 3.6) ensures under-approximation by treating an inter-procedural constraint as `true` when there exists a callee candidate containing unsupported features. This can result in some function templates having trivial constraints.

Summary. Our analysis is highly efficient and can synthesize non-trivial constraints for 30%-40% of function templates. In particular, the high efficiency makes it possible to use our tool in interactive settings where the developer is editing the code.

4.2 Precision on STL Library

To further understand the quality of the synthesized non-trivial constraints for `algorithm`, we conduct a semi-automated comparison of our generated requirements with the ones documented in the C++ standard. We choose the 14 function templates from the introductory C++ textbook [53] as our targets. These function templates are regarded as "particularly useful" [22].

The measurement process is as follows. First, we collect a set of representative *named requirements* [10] that are used in the C++ standard to specify the expectations of template parameters in the standard library. We create a constraint formula evaluator, which incorporates certain named requirements as hard-coded components. The evaluator then runs on the synthesized constraint formula and reports the most general named requirement N that evaluates to true on the formula. This implies that N is at least as constrained as the formula itself. This method can, in general, be used

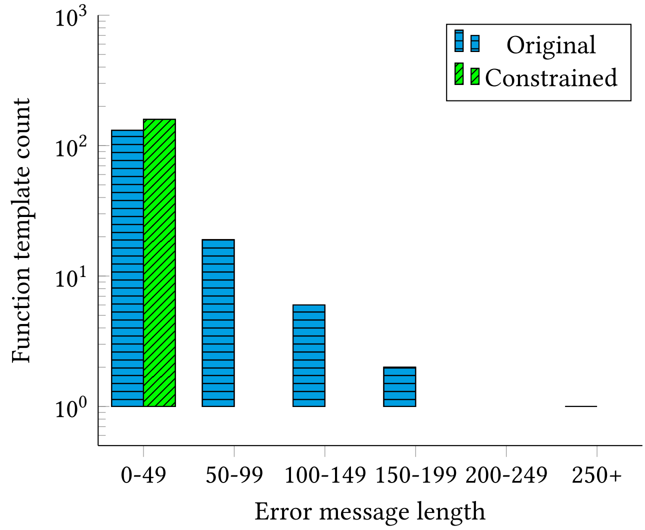


Fig. 11. Error message length (number of lines) distribution on `algorithm`. The y-axis is of logarithm scale, so most lengths reside in $[0, 50)$. The average lengths are 30.022 for the original code and 13.019 for the constrained code.

for matching constraints with pre-defined concepts, as discussed in Section 6.1. Next, we manually compare the generated named requirements with the ones described in the documentation.

Table 2 shows the library requirements obtained from the document and the synthesized requirements obtained from the above process. We can observe that the synthesized requirements always under-approximate the library requirements, and they are often similar or identical to the documented requirements. For the first type template parameter of `for_each`, the library requirement is `InputIterator`, while the generated constraint formula is

```
(requires (T x0) { *x0; } && requires (T x0) { ++x0; })
```

which is inferred to be just `Iterator` by our formula evaluator. The distinction between these types of iterators can be complex [9], involving details that our tool does not handle. However, the conclusion of `Iterator` already provides more information than the unconstrained case, and in clearer cases, our tool can also synthesize more precise iterators such as the `RandomAccessIterator` for `sort`, where the code of `sort` provides enough information for our tool to differentiate it from normal iterators. For `copy` and `move` (not to be confused with the `move` for move semantics), our tool synthesized trivial constraints. This is because in the version of STL targeted by our experiment, they were implemented using a helper `struct _ClassicAlgPolicy`, which our tool currently does not handle.

Summary. Our tool can synthesize meaningful constraints under-approximating the standard library document for `algorithm`, and in many cases the generated constraints are the same as the library requirements. This demonstrates that our tool can synthesize constraints improving the template interface's clarity.

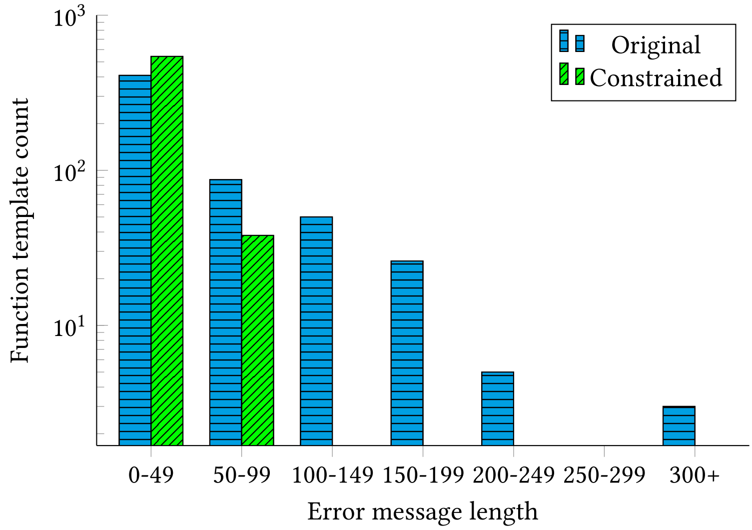


Fig. 12. Error message length (number of lines) distribution on `special_functions`. The y-axis is of logarithm scale, so most lengths reside in $[0, 50)$. The average lengths are 43.769 for the original code and 15.826 for the constrained code.

4.3 Error Message Reduction on STL Library and Boost Library

We also examine the reduction in error messages when incorrect arguments are used for both `algorithm` and `special_functions`. Specifically, for each function template `f`, we use a Python script to introduce a new empty `struct S {};` along with a variable `s` of type `struct S` into the code. We then make a function call `f(s, s, ...)`. Given that the new type `S` is unlikely to satisfy the requirements of `f`, template-related error messages are expected. We compare the lengths (numbers of lines) of error messages before and after the addition of the constraints. Figure 11 and Figure 12 give the distribution of these lengths. Note that the Boost library `special_functions` includes some STL headers as well, and for measurements of the error message reductions in this case, we excluded STL function templates and only considered Boost function templates.

From these two figures, it is clear that the synthesized concepts can successfully intercept the errors at the early stages and greatly reduce the length of error messages. Constrained templates consistently produce error messages that are less than 100 lines for both libraries. This provides strong evidence that the new errors are more comprehensible and manageable.

Since length is not the only measurement of error messages' readability, we also report the number of cases where the added constraint results in an error message containing the string "constraints not satisfied". Among 158 original errors for `algorithm`, 137 constrained errors contain "constraints not satisfied"; among 582 original errors for `special_functions`, 214 constrained errors contain "constraints not satisfied". This provides further evidence that many constrained templates can intercept the error in an early stage of instantiation and improve error message readability. Section 5.3 also presents a concrete case study where the constrained version is not much shorter but is arguably more readable.

Summary. The constraints synthesized by our tool for `algorithm` and `special_functions` can effectively reduce the lengths of error messages. Additionally, even for shorter error messages, the constrained version could be more comprehensible, as discussed in Section 5.3.

```
int elem = 12;
bool isElementPresent = std::binary_search(
    v.begin(), v.end(), elem,
    [](const X& right, const X& left) { return right.attribute_1 < left.attribute_1; }
);
```

(a) Case study 1 code snippet.

```
sort(graph->edge[0], graph->edge[(graph->E)-1], myComp<T>);
```

(b) Case study 2 code snippet.

```
struct S {};
int main() {
    S s;
    boost::math::sign(s);
}
```

(c) Case study 3 code snippet.

Fig. 13. C++ code snippets for case studies.

5 Case Studies

To better understand the error message improvements, we conduct three case studies. In Section 4, our tool synthesized constraints for `<algorithm>` from the Standard Template Library (STL) and `<boost/math/special_functions.hpp>` from the Boost library. We select two incorrect C++ programs using STL and one incorrect C++ program using Boost, and study the error message improvements after adding the synthesized constraints to STL/Boost. The incorrect programs using STL are two real-world examples obtained from the StackOverflow Q&A website, and the incorrect program using Boost is one obtained from our synthetic incorrect programs in Section 4.3. All errors were produced by Apple Clang version 15.0.0.

5.1 Case 1: Real World Error Example of STL Binary Search from StackOverflow

Figure 13a gives a C++ code snippet, which is based on a real-world question from StackOverflow [49]. The question is about a compilation error on C++ code with an incorrect use of `std::binary_search` on a vector of a custom class: the provided lambda expression only accepts objects of the custom class while the target of the binary search is an `int`. According to the C++ specification,[7] the last parameter of `std::binary_search` should be a binary predicate `bool pred(const Type1 &a, const Type2 &b)` such that the object `elem` can be implicitly converted to both `Type1` and `Type2`, and the iterators can be dereferenced and then implicitly converted to both `Type1` and `Type2`. Our tool synthesized constraints for `std::binary_search` so the error message for that code can be improved. The effective part of the added constraint is as follows.

```
requires (_Compare f, _Tp x0, _ForwardIterator x1) { f(x0, *x1); }
```

The error messages before/after adding the constraint are shown in Figure 14. Note that in the error messages before adding the constraint, there are 20 omitted lines, which matches Section 4.3's conclusion that adding constraints can effectively reduce the error message length. Also, in the error messages before adding the constraint, there are unnecessary implementation details such

[7] https://en.cppreference.com/w/cpp/algorithm/binary_search.

```
In file included from test1.cc:1:
In file included from /Library/Developer/CommandLineTools/SDKs/...
In file included from /Library/Developer/CommandLineTools/SDKs/...
...failed due to requirement '__is_callable<(lambda at test1.cc...
  static_assert(__is_callable<_Compare, decltype(*__first), const...
  ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~...
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
    __first = std::lower_bound<_ForwardIterator, _Tp, __comp_ref...
                   ^
test1.cc:16:34: note: in instantiation of function template spec...
    bool isElementPresent = std::binary_search(
                                 ^
In file included from test1.cc:1:
In file included from /Library/Developer/CommandLineTools/SDKs/...
In file included from /Library/Developer/CommandLineTools/SDKs/...
...lower_bound.h:40:9: error: attempt to use a deleted function
    if (std::__invoke(__comp, std::__invoke(__proj, *__m), __...
        ^
...specialization 'std::__lower_bound_impl<std::_ClassicAlgPolicy...
  return std::__lower_bound_impl<_ClassicAlgPolicy>(__first, __...
              ^
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
(20 lines omitted)
3 errors generated.
```

(a) Error messages of the template `std::binary_search` before adding constraints.

```
...:16:29: error: no matching function for call to 'binary_search'
    bool isElementPresent = std::binary_search(
                            ^~~~~~~~~~~~~~~~~~
...candidate template ignored: constraints not satisfied [with...
binary_search(_ForwardIterator __first, _ForwardIterator __last,...
^
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
...requires (_Compare f, _Tp x0, _ForwardIterator x1) { f(x0, *x1); }
                                                          ^
/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk...
binary_search(_ForwardIterator __first, _ForwardIterator __last,...
^
1 error generated.
```

(b) Error messages of the template `std::binary_search` after adding constraints.

Fig. 14. Error messages comparison on `std::binary_search`.

as `std::__lower_bound_impl<_ClassicAlgPolicy>`, while in the error messages after adding the constraint a clear explanation of the compilation error "`constraints not satisfied`." In particular, it shows that the predicate passed to the call does not satisfy the requirements.

5.2 Case 2: Real World Error Example of STL Sorting from StackOverflow

Figure 13b gives a C++ code snippet, which is based on a real-world question from StackOverflow [52]. The question is about a compilation error on the C++ code with the incorrect use of `std::sort` on an array of a custom class. According to the C++ specification,[8] the first and second parameters of `std::sort` should be a pair of iterators. The incorrect code uses a pair of custom objects of **`class Edge`**. Our tool synthesized constraints for `std::sort`. The effective part of the added constraint is as follows.

```
requires
(
 requires (_RandomAccessIterator x0) { *x0; } &&
 requires (_RandomAccessIterator x0) { ++x0; } &&
 requires (_RandomAccessIterator x0) { --x0; }
)
```

[8] https://en.cppreference.com/w/cpp/algorithm/sort.

```
(135 lines omitted)
...: note: in instantiation of...
    std::sort(graph->edge[0], graph->edge[(graph->E)-1], myComp<int>);
         ^
...: error: no type named...
  typedef typename iterator_traits<_RandomAccessIterator>::...
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~...
...: note: in instantiation of...
    std::__sort<_WrappedComp>(std::__unwrap_iter(__first), ...
         ^
...: error: invalid operands...
  difference_type __depth_limit = 2 * __log2i(__last - __first);
                                              ~~~~~~ ^ ~~~~~~~
...: note: candidate template ignored:...
operator-(const reverse_iterator<_Iter1>& __x, const...
^
12 warnings and 8 errors generated.
```

(a) Error messages of the template `std::sort` before adding constraints.

```
(33 lines omitted)
...: warning: user-defined literal suffixes not starting with '_'...
    __attribute__((__visibility__("hidden")))...
...^
...: error: no matching function for call to 'sort'
    std::sort(graph->edge[0], graph->edge[(graph->E)-1], myComp<int>);
    ^~~~~~~~
...: note: candidate template ignored: constraints not satisfied...
void sort(_RandomAccessIterator __first, _RandomAccessIterator __last...
     ^
...: note: because '*x0' would be invalid...
 requires (_RandomAccessIterator x0) { *x0; } &&
                                       ^
...: note: candidate function template not viable:...
void sort(_RandomAccessIterator __first, _RandomAccessIterator __last) {
     ^
12 warnings and 1 error generated.
```

(b) Error messages of the template `std::sort` after adding constraints.

Fig. 15. Error messages comparison on `std::sort`.

The error messages before/after adding the constraint are shown in Figure 15. Before adding the constraints, there are 151 lines of warnings and errors generated (where the warnings are related to C++ preprocessing in our experimental steps and are irrelevant to the main errors), where there are 8 errors in total. After adding the constraints, the number of lines of warnings and errors is reduced to 49, and the number of errors is reduced to 1. Again, the new error message provides a clear explanation of the reason: "`constraints not satisfied`." In particular, the `*x0` in the error message clearly explains the problem: dereferencing an object of type **`class Edge`** is invalid.

5.3 Case 3: Synthetic Error Example of Boost from Our Experiments

Figure 13c gives a C++ code snippet, which is based on one of the synthetic erroneous code described in Section 4.3. The code makes use of `boost::math::sign`, but the argument type **`struct S`** is incorrect, resulting in compilation errors. Our tool synthesized constraints for `boost::math::sign`. The effective part of the added constraint is as follows.

```
requires (T x0, int x1) { x0 == x1; }
```

The error messages before/after adding the constraint are shown in Figure 16. Although, in this case, the lengths of error messages before/after adding the constraint are similar, it is arguably true that the messages after adding the constraint are clearer and easier to understand. Specifically, before adding the constraint, the implementation details of the function template are exposed (`return (z == 0) ? 0 : (boost::math::signbit)(z) ? -1 : 1;`). After adding the constraint, it

```
<stdin>:76087:14: error: invalid operands to binary expression
('const S' and 'int')
   return (z == 0) ? 0 : (boost::math::signbit)(z) ? -1 : 1;
           ~ ^  ~
<stdin>:111864:14: note: in instantiation of function template
specialization 'boost::math::sign<S>' requested here
boost::math::sign(s);
             ^
1 error generated.
```

(a) Error messages of the template `boost::math::sign` before adding constraints.

```
<stdin>:126854:1: error: no matching function for call to 'sign'
boost::math::sign(s);
^~~~~~~~~~~~~~~~~
<stdin>:80713:12: note: candidate template ignored:
constraints not satisfied [with T = S]
inline int sign (const T& z)
           ^
<stdin>:80712:30: note: because 'x0 == x1' would be invalid:
invalid operands to binary expression ('S' and 'int')
requires (T x0, int x1) { x0 == x1; }
                             ^
1 error generated.
```

(b) Error messages of the template `boost::math::sign` after adding constraints.

Fig. 16. Error messages comparison on `boost::math::sign`.

becomes evident that the template is ignored due to "`constraints not satisfied`" and "`because 'x0 == x1' would be invalid`." In particular, the erroneous code causes the comparison between `s` and an **`int`**, which is invalid.

6 Discussions

6.1 Matching with Existing Concepts

Our approach synthesizes a constraint formula for each type template parameter. However, there are also pre-defined concepts such as `std::integral` or domain-specific ones created by programmers. Our approach can be easily extended to match the synthesized constraint formula with these pre-defined concepts as explained in Section 4.2, using evaluators on the constraint formulas. Specifically, we view the pre-defined concept as a predicate P, and suppose the synthesized constraint formula is of the form $((A \vee B) \wedge C)$. We first check whether P implies atomic components A, B, C, and then combine them using $\vee$ or $\wedge$. This idea for comparing predicates is also general and can potentially have other usage scenarios.

6.2 Generalization

Our approach targets `C++` templates, but the general idea is applicable to various programming languages in different usage scenarios. For example, disjunctions can model the requirements of various kinds of polymorphisms, such as overloading, dynamic dispatch, etc. The backmap 3.4 idea can, in general, retain different information as needed at call-sites. Our goal of "synthesizing constraints for `C++` templates" can also be abstracted to "synthesizing requirements for functions," so it shares similarities with API/specification inference techniques [45]. The paper focuses on `C++` as a concrete and tangible illustration of the overarching idea.

6.3 High Level Semantics of Programming Languages

Our work also advocates attention to the high-level semantics of modern programming languages. In particular, `C++` templates themselves are not translated into middle-level IR or assembly code by the compiler: only template instantiations are preserved. Indeed, the main problem we target (compilation error) is no longer accessible in middle-level IR, such as LLVM-IR [34]. To deal with such high-level semantics, it is necessary to confront complicated high-level program representations. In our case, we directly analyze the C++ AST.

6.4 Usage Scenarios

We expect our tool to be used mainly in two scenarios. First, our tool can be applied to existing templated C++ code, which can improve the interface and serve as a precaution for future error messages. Second, because of the high analysis speed (Section 4.1), our tool can be integrated into IDEs (similar to refactoring tools) and thus provide interactive feedback even when the developers are changing the code. In both usage scenarios, the developer can choose to either accept or reject the synthesized constraints to ensure absolute soundness.

6.5 Limitations

Section 3.6 mentions two limitations of our current implementation: (1) in certain cases precision is sacrificed to preserve general correctness, and (2) cv-qualifiers and lvalue/rvalue references are ignored to maintain a lightweight implementation, which may lead to unsoundness for these features. In general, achieving full precision is inherently impossible due to the undecidability discussed in Section 2.2. Furthermore, undecidability implies that it is not possible to automatically tell whether a synthesized trivial constraint (`true`) is genuine. However, static analysis techniques could provide partial insights into this question, potentially serving as a foundation for a more precise approach. Future work could focus on improving precision and incorporating support for qualifiers and references.

7 Related Work

This work attacks a relatively unexplored problem: C++ template constraint synthesis. Our work improves the readability and maintainability of C++ code by leveraging the features of C++20. Our novel technical insight includes using lightweight static analysis to handle complicated programming languages like C++ in real-world scenarios while still ensuring high rigor and soundness (Section 3.6). This section focuses on surveying related topics.

Our work shares similarities with type inference for dynamically-typed languages, such as JavaScript [3] and Python [39, 60]. However, as highlighted in the introduction, our work does not infer types but infers constraints corresponding to sets of types, synthesizes complicated constraints applicable to newly defined types, and can be regarded as a meta-analysis of the compilation process for the statically-typed language C++. There are also previous efforts targeting Java wildcard inference [1, 2].

Existing static analysis work for C++ typically focuses on traditional and general topics such as symbolic execution [29, 30] and model checking [5, 26, 36, 37], which rarely targets high-level C++ semantic problems. In contrast, our work handles recent language features (constraints and concepts) introduced in C++20. While there are existing studies on the formal semantic analysis of C++ templates [46] and comparisons with other languages like Haskell [6], our work presents a practical application for C++ templates. There also exists work handling high-level semantics such as Java reflection [31] and containers [58], while our work focuses on C++ generic programming. There also exists work improving type error messages by using the type checker as an oracle to search for similar programs that do type-check [28]. Our work adds "specifications" to the original programs, which itself is beneficial for a clearer interface even if no error occurs.

Our work also shares similarities with API/specification [45] inference techniques, but we focus on soundy type requirements, while existing API/specification inference techniques [7, 14, 33, 42, 59] utilize data-mining, probabilistic methods, or various heuristics.

The idea of backmap (Section 3) can be regarded as a form of compile-time context-sensitivity, similar to the usual run-time context-sensitivity [15, 24, 44, 54, 61]. In our case, the same function template can be called inside different function templates, each of which passes potentially different "type-contexts" into the callees, which are all resolved in compile time.

The field of program synthesis [13, 18, 21, 47] has achieved great progress in recent years. Our work does not synthesize programs but constraints for template parameters in existing code. Thus, our work targets a different problem and is more scalable (Section 4.1) than typical program synthesis techniques.

8 Conclusion

We propose a framework for automatically synthesizing constraints for `C++` function templates. The synthesized requirements are expressed using `C++20` constraints and concepts, which can significantly improve interface clarity and template-related compilation error messages. Our tool runs extremely fast and thus has the potential to be used in IDE for interactive error reporting. Experimental results show that our tool can synthesize valid constraints for real-world `C++` code from the Standard Template Library and the Boost library.

Acknowledgments

The authors thank the anonymous reviewers for their feedback on earlier drafts of this paper. The work described in this paper was supported, in part, by the United States National Science Foundation (NSF) under grants No. 2114627 and No. 2237440; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

Data-Availability Statement

The implementation of our constraint synthesizer, and the corresponding experimental scripts and data are available on Zenodo [16]. The latest version of our tool is also made publicly available at https://github.com/sdingcn/concept-synthesizer.

References

[1] John Altidor, Shan Shan Huang, and Yannis Smaragdakis. 2011. Taming the wildcards: combining definition- and use-site variance. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011.* ACM, 602–613. doi:10.1145/1993498.1993569

[2] John Altidor and Yannis Smaragdakis. 2014. Refactoring Java generics by inferring wildcards, in practice. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014.* ACM, 271–290. doi:10.1145/2660193.2660203

[3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3586).* Springer, 428–452. doi:10.1007/11531142_19

[4] Wisnu Anggoro and John Torjo. 2015. *Boost. Asio C++ Network Programming.* Packt Publishing Ltd.

[5] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. 2013. DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044).* Springer, 863–868. doi:10.1007/978-3-642-39799-8_60

[6] Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, and Andreas P. Priesnitz. 2008. A comparison of c++ concepts and haskell type classes. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP*

2008, Victoria, BC, Canada, September 20, 2008. ACM, 37–48. doi:10.1145/1411318.1411324

[7] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 760–774. doi:10.1145/3314221.3314648

[8] Cppreference. Accessed in April 2024. Class template argument deduction. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

[9] Cppreference. Accessed in April 2024. CPP named requirements: LegacyInputIterator. https://en.cppreference.com/w/cpp/named_req/InputIterator.

[10] Cppreference. Accessed in April 2024. Named Requirements. https://en.cppreference.com/w/cpp/named_req.

[11] Cppreference. Accessed in April 2024. SFINAE. https://en.cppreference.com/w/cpp/language/sfinae.

[12] Cppreference. Accessed in April 2024. Template argument deduction. https://en.cppreference.com/w/cpp/language/template_argument_deduction.

[13] Loris D'Antoni, Qinheping Hu, Jinwoo Kim, and Thomas W. Reps. 2021. Programmable Program Synthesis. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*. Springer, 84–109. doi:10.1007/978-3-030-81685-8_4

[14] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing deep-learning libraries via automated relational API inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 44–56. doi:10.1145/3540250.3549085

[15] Shuo Ding and Qirun Zhang. 2023. Mutual Refinements of Context-Free Language Reachability. In *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14284)*. Springer, 231–258. doi:10.1007/978-3-031-44245-2_12

[16] Shuo Ding and Qirun Zhang. 2025. *Fast Constraint Synthesis for C++ Function Templates (Artifact)*. doi:10.5281/zenodo.14945421

[17] Lee Zhi Eng. 2016. *Qt5 C++ GUI programming cookbook*. Packt Publishing Ltd.

[18] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 599–612. doi:10.1145/3009837.3009851

[19] Ronald Garcia and Andrew Lumsdaine. 2009. Toward foundations for type-reflective metaprogramming. In *Generative Programming and Component Engineering, 8th International Conference, GPCE 2009, Denver, Colorado, USA, October 4-5, 2009, Proceedings*. ACM, 25–34. doi:10.1145/1621607.1621613

[20] William Gasarch. 1998. A survey of recursive combinatorics. In *Studies in Logic and the Foundations of Mathematics*. Vol. 139. Elsevier, 1041–1176. doi:10.1016/S0049-237X(98)80049-9

[21] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Found. Trends Program. Lang.* 4, 1-2 (2017), 1–119. doi:10.1561/2500000010

[22] ISO. Accessed in January 2025. A Tour of CPP: Containers and Algorithms. https://isocpp.org/files/4-Tour-Algo-draft.pdf, 117 pages.

[23] ISO. Accessed in October 2024. The Standard Definition of CPP. https://www.iso.org/standard/79358.html.

[24] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. doi:10.1145/3498720

[25] Nicolai M Josuttis. 2012. The C++ standard library: a tutorial and reference. (2012).

[26] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (2018), 17:1–17:32. doi:10.1145/3158105

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86. doi:10.1109/CGO.2004.1281665

[28] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for type-error messages. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 425–434. doi:10.1145/1250734.1250783

[29] 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 - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*. Springer, 609–615. doi:10.1007/978-3-642-22110-1_49

[30] Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan. 2012. KIL: An Abstract Intermediate Language for Symbolic Execution and Test Generation of C++ Programs. Citeseer, 15.

[31] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* 28, 2 (2019), 7:1–7:50. doi:10.1145/3295739

[32] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46. doi:10.1145/2644805

[33] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. ACM, 75–86. doi:10.1145/1542476.1542485

[34] LLVM. Accessed in April 2024. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[35] Edward J McCluskey. 1956. Minimization of Boolean functions. *The Bell System Technical Journal* 35, 6 (1956), 1417–1444. doi:10.1002/j.1538-7305.1956.tb03835.x

[36] Felipe R. Monteiro, Mikhail R. Gadelha, and Lucas C. Cordeiro. 2022. Model checking C++ programs. *Softw. Test. Verification Reliab.* 32, 1 (2022). doi:10.1002/STVR.1793

[37] Brian Norris and Brian Demsky. 2016. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.* 38, 3 (2016), 10:1–10:51. doi:10.1145/2806886

[38] Bo Pang, Erik Nijkamp, and Ying Nian Wu. 2020. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics* 45, 2 (2020), 227–248.

[39] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael R. Lyu. 2022. Static Inference Meets Deep learning: A Hybrid Type Inference Approach for Python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2019–2030. doi:10.1145/3510003.3510038

[40] Willard V Quine. 1952. The problem of simplifying truth functions. *The American mathematical monthly* 59, 8 (1952), 521–531. doi:10.2307/2308219

[41] Willard V Quine. 1955. A way to simplify truth functions. *The American mathematical monthly* 62, 9 (1955), 627–631. doi:10.1080/00029890.1955.11988710

[42] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 123–134. doi:10.1145/1250734.1250749

[43] Gabriel Dos Reis and Bjarne Stroustrup. 2006. Specifying C++ concepts. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. ACM, 295–308. doi:10.1145/1111037.1111064

[44] Thomas W. Reps. 2000. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 162–186. doi:10.1145/345099.345137

[45] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Trans. Software Eng.* 39, 5 (2013), 613–637. doi:10.1109/TSE.2012.63

[46] Jeremy G. Siek and Walid Taha. 2006. A Semantic Analysis of C++ Templates. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4067)*. Springer, 304–327. doi:10.1007/11785477_19

[47] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM, 313–326. doi:10.1145/1706299.1706337

[48] StackExchange. Accessed in April 2024. Generate the longest error message in CPP. https://codegolf.stackexchange.com/a/10470.

[49] StackOverflow. Accessed in April 2024. Compilation error for a binary search on the attributes. https://stackoverflow.com/questions/42604796/compilation-error-for-a-binary-search-on-the-attributes.

[50] StackOverflow. Accessed in April 2024. Deciphering CPP template error messages. https://stackoverflow.com/questions/47980/deciphering-c-template-error-messages.

[51] StackOverflow. Accessed in April 2024. How to improve compiler error messages when using CPP std::visit? https://stackoverflow.com/questions/72507596/how-to-improve-compiler-error-messages-when-using-c-stdvisit.

[52] StackOverflow. Accessed in October 2024. Using stdsort in CPP with iterators and templates. https://stackoverflow.com/questions/59096297/using-stdsort-in-c-with-iterators-and-templates.

[53] Bjarne Stroustrup. 2022. *A Tour of C++*. Addison-Wesley Professional.

[54] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. doi:10.1145/3485524

[55] Tumblr. Accessed in April 2024. The Grand CPP Error Explosion Competition. https://tgceec.tumblr.com/.

[56] Christopher Umans, Tiziano Villa, and Alberto L Sangiovanni-Vincentelli. 2006. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7 (2006), 1230–1246. doi:10.1109/TCAD.2005.855944

[57] Todd L Veldhuizen. 2003. *C++ templates are Turing complete*. Technical Report. Indiana University.

[58] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. 2022. Complexity-guided container replacement synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–31. doi:10.1145/3527312

[59] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 335–346. doi:10.1109/ICPC.2019.00052

[60] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. ACM, 607–618. doi:10.1145/2950290.2950343

[61] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 344–358. doi:10.1145/3009837.3009848

Received 2024-10-08; accepted 2025-02-18