Source Language Representation of Function Summaries in Static Analysis

Gábor Horváth
Department of Programming Languages and Compilers
Eötvös Loránd University, Faculty of Informatics
Pázmány Péter sétány 1/C
Budapest, Hungary
xazax@caesar.elte.hu

Norbert Pataki
Department of Programming Languages and Compilers
Eötvös Loránd University, Faculty of Informatics
Pázmány Péter sétány 1/C
Budapest, Hungary
patakino@caesar.elte.hu

ABSTRACT

Static analysis is a popular method to find bugs. In context-sensitive static analysis the analyzer considers the calling context when evaluating a function call. This approach makes it possible to find bugs that span across multiple functions. In order to find those issues the analyzer engine requires information about both the calling context and the callee. Unfortunately the implementation of the callee might only be available in a separate translation unit or module. In these scenarios the analyzer either makes some assumptions about the behavior of the callee (which may be unsound) or conservatively creates a program state that marks every value that might be affected by this function call. In this case the marked value becomes unknown which implies significant loss of precision.

In order to mitigate this overapproximation, a common approach is to assign a summary to some of the functions, and each time the implementation is not available, use the summary to analyze the effect of the function call. These summaries are in fact approximations of the function implementations that can be used to model some behavior of the called functions in a given context. The most proper way to represent summaries, however, remains an open question.

This paper describes a method for summarising C (or C++) functions in C (or C++) itself. We evaluate the advantages and disadvantages of this approach. It is challenging to use source language representation efficiently due to the compilation model of C/C++. We propose an efficient solution. The emphasis of the paper is on using static analysis to find errors in the programs, however the same approach can be used to optimize programs or any other tasks that static analysis is capable of. Our proof of concept implementation is available in the upstream version of the Clang compiler.

ICOOOLPS '16, July 18 2016, Rome, Italy

DOI: http://dx.doi.org/10.1145/3012408.3012414

CCS Concepts

•**Theory of computation** → **Program analysis;** *Pre- and post-conditions; Parsing;*

Keywords

C++, static analysis, Clang, function summaries

1. INTRODUCTION

Static analysis is the analysis of a program without executing it. This analysis is carried out by an automated tool.

The maintenance costs of the software are increasing with the size of the codebase. The more tasks that can be automated the better. Static analysis has great impact on reducing maintenance costs of complex software [12]. For example, compilers can detect more optimization possibilities statically and these optimizations make it possible to develop software using high level language features. Static analysis, however, is also a great tool for finding bugs and code smells [5]. The earlier a bug is detected, the lower the cost of the fix is. This makes static analysis a useful and cheap supplement to testing. Rigorous static analysis can be used to verify the correctness of programs or detect security vulnerabilities, making it essential for safety critical systems. The main focus of the paper is to use the presented approach in a bug finding tool, but it is possible to leverage these ideas in other related areas as well.

Some of the errors that can be detected by static analysis tools span across multiple subprograms. For example, the same object might be deleted in two distinct procedures. To find such errors some tools implement context-sensitive interprocedural analysis. One of the challenges with this approach is that the implementation of a function might only be defined in a separate translation unit from the one that is being analyzed. In these scenarios the analyzer either makes some assumptions about the behavior of the callee (which may be unsound) or conservatively creates a program state that marks every value that might be affected by the procedure call. The marked value becomes unknown which implies significant loss of precision. The naive solution would be to look up the implementation of those procedures from the other translation units. This is only partial solution. The implementation might only be available in a separate module or library. The source code of the library which contains the implementation might not be available

for the developers. Moreover, this approach might involve significant amounts of duplicated work in the C compilation model.

A common approach to mitigate this problem is to assign a summary to some of the functions. Each time the implementation is not available, the summary is used to analyze the effect of the function call. These summaries are in fact approximations of the function implementations that can be used to model some behavior of the called functions in a given context. Those summaries can be synthesized from the source code during a multi pass analysis or written manually by a developer.

This paper describes a method for summarising C (or C++) functions in C (or C++) itself. We evaluate the advantages and disadvantages of this approach. The compilation model of C/C++ makes it very challenging to use same language representation efficiently. We propose an efficient solution. Our proof of concept implementation is available in the upstream version of the Clang compiler [11]. C++ is a very popular target for static analysis tools due to the error-prone nature of several language features [16].

This paper is organized as follows. In section 2 related work is discussed: similar approaches, tools and language constructs are presented. In section 3 the challenges of analyzing of C/C++ are detailed. The Clang compiler infrastructure is introduced in section 4. The notation of summaries, the pros and cons are presented in section 5. Section 6 describes some methods to synthesize summaries. Section 7 describes the implementation details of our Clang-based tool. We define some directions of further work in section 8. Finally, this paper concludes in section 9.

2. RELATED WORK

Summaries are defined in the dragon book [1] as a tool for interprocedural analysis. A notable usage of function summaries for C programs is in Andersen's implementation of points to analysis [4]. Large programs, however, are typically built using separate compilation. Traditional whole-program analysis cannot be used. Rountev and Ryder generalized Andersen's analysis to work in a modular context [15]. This work is, however, specific to a single type of analysis. The summaries introduced in this paper are suitable for abstract interpretation techniques in general.

Ali and Lothak developed a method to create a sound call graph for the application part of a program without analyzing the code of the libraries [2]. This method is useful since many analyses depend on a call graph. They have created a tool to generate a library that can be used with the existing whole program analysis frameworks. This library is in fact a collection of summaries [3]. This generated library is made of Java bytecode. Thus, their method can also be considered as a same language representation of summaries. They also discuss the advantages of the same language representation, namely existing tools can rely on the output of their tool without any modification. The automatic generation of the library is similar to the approach that we propose to synthesize the function summaries. The nature of the summaries we discuss in this paper, however, is different, as it is created to aid a broad range of analyses.

Gopan and Reps managed to do a summarization of libraries based on the low level binaries of the libraries [6]. Unfortunately, during the compilation process, the high level type information is lost. Summaries based on low level representation are not suitable for analyses that depend on high level type information. Our proposed solution relies on high level summaries.

```
S (x);
T * y;
func((R) * z);
T T;
```

Figure 1: Ambiguous C grammar

3. THE C/C++ COMPILATION MODEL

In this chapter we discuss the difficulties of analyzing C and C++ source code. Unfortunately, these languages are very hard to analyze. One of the biggest obstacles is the compilation model.

Compilation time and tooling support depends on language design. The grammar and the type system of a language can have significant effect on the compilation time and the complexity of language related tools. The compilation model has similar impacts.

The grammar of C and C++ is ambiguous without sufficient type information. There are some examples of ambiguity in Figure 1. If S is a type, then the first line is the definition of a variable, otherwise, if S is a function, then that expression is a function call. The second line can be interpreted as a definition of a pointer or a multiplication depending on the type context. The third line can be a multiplication or a dereference followed by a C type cast. The last line shows that it is possible to have the same name as both a variable name and a type name in the same scope. To get the type information the headers included in a file need to be parsed before the file can be parsed.

C++ is notorious for slow parsing times [13], and this phenomenon also degrades the quality of tooling. There are header files which are textually copied into the translation units during compilation. The problem is that a header file might be included in plenty of the translation units, so the compiler has to reparse that chunk of code each time. It is not possible to avoid redundant parsing of the headers in general (by caching intermediate representations), due to the fact that they might be parsed in a different preprocessor state each time.

There are a few workarounds for this problem, but none of them is perfect. One can cache the intermediate representation of some subset of the headers. The other workaround is to do a unity build [13]. Unity build is an approach, where the whole project consists of only a single translation unit. This way every header file will only be parsed once. Unfortunately it is not easy to achieve a unity build. The naive solution of including all the files into a single translation unit might cause semantic changes to the code. Unity builds also have scalability problems.

Note that, the slow parsing time is an inherent problem of the compilation model and not a compiler or static analysis tool dependent implementation detail. There is a combinatorial explosion in how many times headers need to be reparsed. Every C and C++ related tool needs to deal with this problem.

Summaries are a common way of approaching whole program analysis in the context of separate compilation. Due to the complexity of static analysis, it is natural to strive for

```
int i;

if (a) {
  i = 0;
} else {
  i = 1;
}
...
if (!a) {
  j = 5/i;
} else {
  j = 3/i;
}
j += 2/i;
```

Figure 2: Example for division by zero

methods that can be used with existing unmodified tools or require only minimal modifications. Same language representation of summaries is a way to do this. The representation does not need to be a textual, a token stream, a syntax tree, or any other intermediate representation can still be considered as same language representation.

Unfortunately, the compilation model makes it hard to use summaries that are represented in C or C++. In case the representation is textual, the headers that contain the declarations need to be included in the summaries. That makes the summaries very heavy weight and inefficient to use. In case the representation is an Abstract Syntax Tree (AST) we do not need to parse anything. The definitions that are used in the body of a summary are not guaranteed to be available in the translation unit where the summary is used. In order to be able to use the AST of the summary it must contain all the necessary type definitions. This makes the AST representation heavy weight as well. Moreover, in both cases, the type context of the function body and the analyzed translation unit need to be merged. Extracting the function bodies from the LLVM codebase (2M+ LOC) along with the necessary type definitions occupies more than 45 GB.

4. CLANG AND STATIC ANALYZER

Clang is an open source compiler for C/C++/Objective-C. It is built on the top of the LLVM compiler infrastructure. It is a rapidly evolving project which is supported by Apple, Google, and Microsoft.

One of the advantages of Clang is its modular architecture. One can use the parser as a library to build tools. The Static Analyzer is the part of the Clang compiler. It is utilizing the same AST and CFG that is generated by the compiler.

Clang supports several levels of analysis. It can do pattern matching on the AST. In C and C++ the parsing and the semantic analysis can not be separated due to the fact that the ambiguities in the grammar can only resolved using the type information. This implies that the AST produced by Clang always contains the type information. Clang has an embedded domain specific language called ASTMatchers that can be used for certain types of analysis [10].

Clang can also do flow-sensitive analysis using the Control Flow Graph (CFG). In Clang, however, this method is rarely used due to its limitations. Consider an analysis that checks for division by zero errors. It would track the values of the variables and store which ones are zero. What should such analysis do, when two branches fold together? One option would be to mark the value of a variable zero if it was zero on any of the branches. If you look at Figure 2, there will be a false positive case when 5 is divided by i, since this branch is only taken when i is not zero. This decision makes this analysis an overapproximation. The other option would be to mark the variable as zero when it is marked as zero on both of the branches. This analysis marks none of the variables as zero in 2. There is a false negative in the last line of the code snippet in this case, this makes the analysis an underapproximation.

The Clang Static Analyzer uses a path-sensitive analysis to avoid these issues. Clang tries to consider each possible execution path of the program. This method has great precision, since the analysis can exclude some impossible branches and it can track the constraints and the values of the variables more precisely. There is a cost of this precision in the execution time: this analysis has exponential complexity in the number of branches. Fortunately the runtime of such analysis turns out to be manageable in practice using various heuristics. Clang Static Analyzer is using this approach on industrial codebases. The analysis time about an order of magnitude bigger than the compilation time. To achieve that, the analyzer does only very limited analysis on the loops and it gives up analyzing too long paths. The Clang Static Analyzer tracks unknown values as symbols and does symbolic computation with them, it is called symbolic execution [8]. The constraints on those symbols are accumulated for each of the paths. There is a constraint manager, that can handle, and in some cases solve these constraints. The biggest limitation of this constraint manager is that it only works with integral values and pointers and it cannot track the relations between two symbols, only the relations between a symbol and a constant. The consequence of this limitation is that the possible values of a symbol is constrained by disjunctions of intervals.

The Static Analyzer does interprocedural analysis. The analysis is also context sensitive. This is implemented using a cloning based approach: the callee's body is "inlined" to the call site each time, when it is called in a new context. This is precise, however, it is also not very efficient. The just "inlined" code might contain more function calls. This can be problematic for recursive functions. In order to avoid infinite recursion, the analyzer has a limit on the size of the call stack. Once such limit is reached, the function calls are evaluated conservatively.

Clang can only analyze one translation unit at a time.

The Static Analyzer is a framework that the developers can extend with custom checkers to carry out specific analysis or model some aspects of the source code. During the analysis, the checkers and the core of the Static Analyzer cooperatively build the so-called exploded graph. For the definition and detailed description of exploded graphs see [14]. The nodes of the exploded graph are representations of program states and program points that are bundled together, the (directed) edges are the transitions between those nodes. In some cases the program execution might not have meaning after a certain node (e.g.: after a division by zero). Those states are sinks in the exploded graph and the analysis stops on that particular path.

The relations between memory regions are also modeled by the Static Analyzer [17]. There is a memory region to

```
void h(int x) { /* ... */ }
void f(int &x);

void g(int x) {
        if (x > 0) {
                h(x);
                f(x);
                h(x);
        }
}
```

Figure 3: Loss of precision

represent the heap, the stack, the global memory and unknown, when the analyzer has no knowledge where a variable might be allocated. There is a memory region for each object, which might be a subregion of another memory region. For example, an element of an array is the subregion of the region of the whole array. Due to the fact that the analyzer does not support global analysis, it cannot reason about the shape of the heap or the values of global variables.

Let us look at Figure 3. The definition of f is in a separate translation unit, the definition of h, however, is available. When the analyzer evaluates g it creates a symbol to represent the unknown value of x. It collects the constraints on this symbol as disjunctions of intervals. It knows that at the point h is called for the first time the value of x is positive due to the constraint in the `if` statement. The value of x after the call to f is unknown, because f might modify it. The consequence is that we have less amount of information about the program state at the second call to h. This loss of precision can be quite severe. In case an element of an array or a member of a struct is passed to f, the whole array or struct is marked as unknown, because f might use pointer arithmetic to modify elements other then the one that was passed to it.

This loss of precision can cause both false positive and false negative results. For example the analyzer might not catch a division by zero error, because the information about the variable which has zero value on that path might be erased by a call to an unknown function. The false positives are the result of analyzing more impossible branches due to the lack of information.

In this paper we consider summaries that reduce the effect of overapproximation. The goal of these summaries is to increase the precision of symbolic execution in general and not to assist a specific check like division by zero or double free. Using same language representation, the summaries do not need to interact with the data structures of the Static Analyzer directly, the analyzer engine can build the required data structures by interpreting the summary the same way it does with other code. This gives us flexibility, the representation of the constraints can be changed without modifying the summaries.

5. SUMMARIES

A summary approximates some behavior of a function. They generally convey less amount of information than the implementation of the function except for the extreme case when an actual implementation itself is the summary. There are several reasons to use summaries:

- Evaluating a summary instead of the function body might be more efficient.
- Summaries can be stored, parsed and looked up more efficiently.
- Summaries can model unsupported language features.

The efficiency of summaries is the result that they tend to be more lightweight (carry less information) then the implementations of the functions. The lookup performance is better because summaries can be indexed and this index can be maintained while they are created or modified. The ability to model unsupported language features is a very interesting property. Only very few symbolic execution engines can reason about bit-level manipulations. When the implementation of a function involves such operations the analyzer cannot evaluate that function properly. The equivalent behavior, however, can be achieved using arithmetic operations. As a consequence the symbolic execution engine can model the effect of the function call better in case a summary is available which is the rewritten version of the original function without bit-wise operations. The reason why the user does not want to write the function without these operations in the first place might be performance or brevity.

Context-insensitive summaries need to refer to the information that is available in the calling context. This can be done using alpha renaming. In each context, the symbols of the summary are renamed to reflect the relationships between the context and the called function. This way the summary can be expressed in terms of the formal parameters, but it will be evaluated each time in terms of the actual parameters. A context-sensitive summary already contains the context-related information within the summary. No extra processing is required when it is applied to the same context again.

One crucial question is, how to get these summaries. One of the possibilities would be to make the developers write them. This approach in general is not feasible due to the large amount of work to create and maintain additional software. It is feasible, however in some circumstances. For example the standard libraries of a language are widely used by the community which makes writing summaries for those libraries a good investment. Summaries created by developers can be better approximations of functions than the automatically synthesized ones. Developers may know how to work around some limitations of the analyzer.

Alternatively, one could define some rules how to infer the summaries from the definitions of the functions. The functions from which we are inferring the summaries in fact can call other functions. The inference rules might take advantage of the information that is available in the summaries that are already inferred. This way the result of the inference depends on which the order the functions are considered. Moreover, the reiteration of the whole procedure might refine the summaries that are generated.

How to represent these summaries? The most suitable representation might be very different for each kind of analysis. This paper focuses on summaries for symbolic execution. During symbolic execution the constraints are recorded for the symbols on each path. The program states can be represented using the language of these constraints. A natural representation of summaries can be seen in Figure 4. Such representation can be used the following way:

$$
\begin{aligned}
&(PreState_1 \implies PostState_{1,1} \vee \cdots \vee PostState_{1,N_1}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
&(PreState_2 \implies PostState_{2,1} \vee \cdots \vee PostState_{2,N_2}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots
\end{aligned}
$$

Figure 4: State based summary representation

```
int strcmp(char * str1,char * str2) {
  int __strcmp(char * str1,char * str2);
  int x = __strcmp(str1, str2);
  if (x > 0) return x;
  if (x < 0) return x;
  return 0;
}
```

Figure 5: Modeling ranges

- Alpha-rename the symbols to match the calling context.
- Look up which *PreState* matches the context.
- For each *PostState* create a new execution path that will be analyzed.
- In case one of the *PostState*s is an error state, report the issue.

The advantage of this representation is that it can be efficient to evaluate the summaries. There are also several disadvantages. It is not easy to represent domain specific information in this constraint language (e.g.: state of streams). It is a new language to learn for developers who want to either check or create summaries. It also lacks extensibility: each time a new analysis requires new kind of information, the constraint language needs to be extended, the summaries need to be regenerated. The summaries also need to be regenerated each time the constraint language evolves (e.g.: when floating point support added) or the representation of the language changes.

Another way to represent summaries is to write them in the same language as the one that is being analyzed. Such summaries can be used together with the cloning approach for interprocedural analysis. There is also no need to implement separate alpha renaming support, the proper modeling of function arguments will solve the problem of context sensitivity in summaries. Since C/C++ is Turing complete, those summaries should be able to model any behavior that is computable. One of the advantages of this approach is that this representation is easy to write and read by developers, there is no need to learn a new language. The fact that analyzer already has the capability to interpret the language makes the implementation easier. The summaries can take advantage of every symbolic execution engine improvement without modifying the code of the summaries. The representation of the constraints can be changed without any modification to the summaries.

To show the expressiveness of this method, the constraint based representation can be rewritten in the analyzed language as shown in Figure 6. Note that the representation of the summary is orthogonal to the fact whether it was written by a developer or inferred from an implementation.

The major disadvantage of the source representation of summaries is performance. The compiler needs to be able to parse the summaries. In order to resolve the ambiguities during parsing C/C++ the type information is required. The type information can be gathered by parsing the headers files that contains the necessary declarations. This is an inherent problem with the compilation model of the language.

One possible solution to this problem is to have hardcoded ASTs as summaries. This way no parsing is needed, however, we lose most of the advantages of the same language representation. Summaries can be edited and maintained without internal knowledge of the compiler. The AST is an internal representation which changes over time, this implies maintenance burden. Changing the summaries requires the recompilation of the compiler which degrades flexibility. This approach was implemented in Clang earlier.

Our proposed solution is to parse these summaries without parsing the required headers. In order to resolve the ambiguities during the parsing, the analyzer can use the type context that is available at the call site. This approach solves the problem of the compilation model of C and C++ for applying summaries. Unfortunately, the compilers are not designed to parse incomplete translation units (that lack the necessary declarations). We needed to modify the parsing procedure of Clang significantly to achieve this functionality. The implementation details are discussed in section 7. In case the summaries are represented using an intermediate representation like AST, only the subtree representing the function body needs to be stored. The types will be resolved from the call site's type context. We also do not need to do type context merging on applying a summary. All in all, using this method the summaries can be lightweight.

In general, the type context at the call site and the type context at the implementation of a function can be quite different. But in C/C++ it is guaranteed that some types used in the function, for example the types of the arguments and the return type is available at the call site. This type context is sufficient to broad range of summaries. It is possible to write the summaries for most of the C standard library functions. Using the caller's type context also implies a limitation, a summary cannot reference any type. This limitation makes it impossible in general to use the implementation itself of a function as a summary. There might be types in the callee's type context that are not available in the caller's context. We have implemented it in the Clang compiler to prove the feasibility of this approach.

Now that we have an efficient way to represent summaries in the analyzed language let us look at how to express different behaviors of the modeled functions.

One of the features of C is that it is possible to declare functions within a function. Our approach takes advantage of this feature during the creation of summaries. Each time a summary wants to represent an unknown value, the summary is able to call a function which has been declared (but not defined) inside the summary just before the function call.

In Figure 5 the *strcmp* function is modeled in a way that forces the symbolic execution engine to consider three paths after each call to the *strcmp* function: the return value is

```
int f(T1 arg1, ... , TN argN) {
  int __getUnknownSymbol();
  assert (Preconditon(arg1, ..., argN));
  if (Pre1(arg1, ..., argN)) {
    switch(__getUnknownSymbol()) {
      case 1: return Post1_1(arg1, ..., argN);
      case 2: return Post1_2(arg1, ..., argN);
      ...
    }
  }
  ...
}
```

Figure 6: Constraint based summary

```
void f(int &x) {
  x += 2;
}
```

Figure 7: Simple summary

positive, the return value is negative, and the return value is zero. The pattern is to first get a value that has an unknown value. Afterwards in case the value is within the range, the summary just returns the value.

There is a summary in Figure 6 that is similar to the state based summary in Figure 4. It contains a branch for each state of the context that needs to be handled separately. In each branch the path is split based on an unknown value. On each path the summary creates the required post state. The preconditions of the function can also be checked. For example in case of square root the argument cannot be negative. Note that, this summary is still context-insensitive since the content of the summary does not depend on the context.

Let us recall the information loss that was demonstrated in Figure 3. Given that we have a hand written summary like in Figure 7, we will not lose any information about the value of x. Moreover, we will have more precise information about its possible value, we will know that its value after the call to f must be greater than 2.

Unfortunately there are behavioral properties that cannot be approximated using this approach. For example in case of a factory function the dynamic type of the returned object might be unknown at compile time. The summary cannot model the dynamic type of the returned object, because that type might not be available in the type context at the call site.

It is possible to automatically generate summaries to model some behavior of the functions. For example, in case we are interested in null pointer dereferences, each time a function dereferences one of its arguments unconditionally, we can generate a summary that has a precondition that the given argument is never null.

6. SYNTHESIS OF SUMMARIES

In a large software project it is not feasible to write and maintain summaries for every important function. In the worst case this would double the work that the developers need to do and potentially introduce false positives due to defects in the summaries.

A better approach would be to synthesize the summaries automatically whenever possible and only write some of them by hand, when the synthesized one is not good enough for some reason.

The natural representation of synthesized summaries in our proposed solution would be the AST. So both the input and the output of the synthesis would be ASTs. There are two possible approaches to do that. The first is to let each check to participate in the synthesis and create pieces of AST and in the end combine those ASTs into one summary. The other approach would be to start with the original AST and replace all the code that employs forbidden types with the proper invalidation.

Unfortunately it is not a trivial problem to let the checkers create arbitrary piece of AST and merge them into a single summary in a sound way. It is possible to come up with a more restricted way to make checkers participate in the summary creation. However, each time this participation scheme is updated the checkers needs to be altered. Moreover, each time a new checker is introduced the summaries need to be regenerated. For these reasons we suggest the other approach: start with the original AST and prune the forbidden parts.

In the summary one can only use those types that are guaranteed to be available in the calling context:

- Primitive types
- Types of the parameters
- Type of the return value
- In case of methods the types of the parent classes

Each user-defined type has its own set of required types:

- The types of the members
- The types of the base classes
- The parameter types of the methods
- The return types of the methods

So the transitive closure of the required types is permitted types that can appear in a summary. Each subexpression that depends on a non-permitted type needs to be replaced for marking the elimination to the analyzer.

The result of this process is illustrated in Figure 8. The original code contains the type `D` which is not permitted in this summary. For this reason the call to `g` and the value of `d` needs to be eliminated. In order to not to derive unsound conclusions, the possible unknown mutation of `c` and the unknown value of `a` is modeled by the summary. These calls are evaluated conservatively by the analyzer.

Notice that during the process of pruning non permitted types from the summary a call to `g` is lost. This means during the synthesis of summaries there might be some cuts to the call chain.

All in all this method is much more precise than evaluating conservatively the cross translation unit calls. It is not as precise as retrieving the original AST but it should perform much better. The speed-up of our method comes from the elimination of redundant header file parsing or redundant storage of header file ASTs.

```
A f(B b, C c) {
  D d;
  if (b.m())
    return c.m();
  A a = g(d, &c);
  return a;
}
```

(a) Original code

```
A f(B b, C c) {
  if (b.m())
    return c.m();
  mutate_unknown(c);
  A a = get_unknown();
  return a;
}
```

(b) Summary

Figure 8: Forbidden type elimination

7. IMPLEMENTATION DETAILS

Parsing of function summaries and utilizing them during the analysis is implemented in the Clang compiler. The implementation is available from version 3.6 [9]. The implementation calls files that consist of summaries *model files*.

In Clang, every parsing related tool is implemented as a `FrontendAction`. The Static Analyzer itself is a `FrontendAction` too. The parsing of model files is carried out by another `FrontendAction` called `ParseModelFileAction`. These actions are responsible for creating an `ASTConsumer` to process the result of the parsing. `ParseModelFileAction` will create a `ModelConsumer`. This consumer will memoize summaries for better performance. The class diagram can be seen in Figure 9.

The static analyzer assigns an instance of `AnalysisDeclContext` to each function declaration. This object handles function-related data such as the AST and the CFG of the body. When a function does not have a body in the processed translation unit the `AnalysisDeclContext` queries `BodyFarm` for a hardcoded AST. We extended `BodyFarm` to look for a model file each time it has no hardcoded implementation for the queried function. The process of creating the AST to use is handled by the `CodeInjector` class. `CodeInjector` is intended to provide a general facility to use an external information source to model a function. The `ModelInjector` subclass of `CodeInjector` uses model files as source. It is essential to keep track of the source of information within `BodyFarm`. Hand-written ASTs do not have associated source code, so they cannot be used in the warning message generation. ASTs coming from model files, however, can be used for sophisticated error reporting.

It is important to parse model files lazily. If the body of the function is not needed it is redundant to load the associated model file. Moreover, to avoid excessive parsing these summaries are cached, so in case the same function is called multiple times the parsing will be done once per translation unit. In Clang the preprocessor owns both the symbol table and the lexer. For this reason the parsing of the model files reuses the same preprocessor object that was used by the analyzer. The type information is stored in the `ASTContext`. This object will be reused as well to avoid ambiguities. The `ParseModelFileAction` will create a new `CompilerInstance` object which is an umbrella object that holds several components of the compiler (e.g.: the preprocessor). During a regular frontend action the `CompilerInstance` creates its own preprocessor and `ASTContext` and releases them after the parsing is done. The ownership model of the `CompilerInstance` had to be changed to support reusing the old ones.

The `Preprocessor` class was not designed to be reused for multiple translation units. One of the biggest challenge was to adapt this class to the new behavior. The file which was given to the invocation of the compiler is called $MainFile$. At the beginning of the parsing process the preprocessor will open the MainFile. In order to parse the model file instead of reparsing the same translation unit this MainFile needs to be set to the model file. The other problem is that there are several predefined identifiers and macros that will be added to the translation unit before it will be parsed. This phase of the compilation needs to be skipped when parsing model files to avoid duplicates. The `SourceManager` also needs to be reused to be able to refer to both source locations in the original translation unit and in the summaries.

The mapping between the function declaration and the name of the model file is based on function name. This mapping is not sufficient to support namespaces and overloading. It is planned to implement a declarative format to make the mapping explicit in those cases. Each file contains a single summary at this time but this can be easily changed. Note that it is not required to have support for templates within summaries because the definitions for template code should always be available in the translation units where it is used.

8. FUTURE WORK

The implementation does not support function overloading, methods and namespaces. Those features, however, are straightforward to add. We also plan to support several summary paths, similar to how several include paths supported for header files. Once they are done we are planning to do some measurements how those summaries affect the performance of the analysis.

We also interested in extending our technique to other languages, like Swift. Swift is a programming language developed by Apple. The compiler has an LLVM backend and it is open source. One of the interesting traits of the Swift compiler is that it lowers the code to an intermediate representation called the Swift Intermediate Language (SIL) [7]. SIL is similar to LLVM intermediate representation, but it has the same type system as Swift. This format is serializable, so it is easy to read and write codes in SIL. This representation is also great to run analysis passes on. We plan to consider such intermediate language as a possible representation of summaries. Our expectation is that it has most of the advantages of our approach but it might have slightly better performance.

We also plan to investigate how suitable the SIL format is to be generated automatically by inference rules.

9. CONCLUSION

We have introduced a new approach to create and maintain summaries for C and C++. This approach involves writing the summary in the same language that is being an-

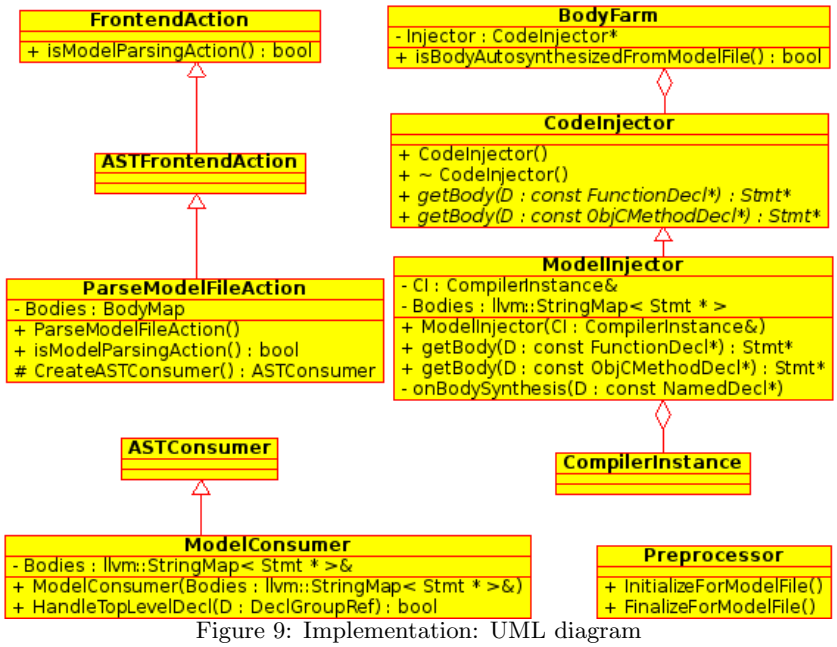

Figure 9: Implementation: UML diagram

alyzed. There are several advantages of this method and it is feasible to implement. An implementation has been introduced and it is part of a mainstream compiler.

The advantages include that the developers do not need to learn a new language to revise or create summaries. It is also easier to maintain summaries represented this way because it is less likely that a modification is needed after a change in the analyzer engine. Unfortunately the naive method to implement textual summaries in C/C++ has severe performance problems. We introduced a method that can solve this problem. We also proposed a way to synthesize summaries that works well with this method automatically.

10. ACKNOWLEDGMENTS

We would like to thank Ted Kremenek the help designing the proof of concept implementation.

11. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] K. Ali and O. Lhoták. Application-only call graph construction. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 688–712, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 378–400, Berlin, Heidelberg, 2013. Springer-Verlag.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[6] D. Gopan and T. Reps. Low-level library analysis and summarization. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 68–81, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] J. Groff and C. Lattner. Swift's high-level ir: A case study of complementing llvm ir with language-specific optimization. Lecture at The ninth meeting of LLVM Developers and Users, 2015.

[8] H. Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 31(1):52–58, Sept. 2005.

[9] G. Horváth. Patch contributed to Clang. http://reviews.llvm.org/rL216550, 2014.

[10] G. Horváth and N. Pataki. Clang matchers for verified usage of the C++ Standard Template Library. *Annales Mathematicae et Informaticae*, 44:99–109, 2015.

[11] C. Lattner. Llvm and clang: Next generation compiler technology. Lecture at BSD Conference 2008, 2008.

[12] Z. Michael and K. C. Robert. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

[13] J. Mihalicza. How #includes affect build time in large systems. In *Proceedings of the 8th international conference on applied informatics (ICAI 2010)*, volume 2, pages 343–350, 2012.

[14] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[15] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries.

In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 20–36, London, UK, UK, 2001. Springer-Verlag.

[16] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[17] Z. Xu, T. Kremenek, and J. Zhang. A memory model for static analysis of C programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, pages 535–548, Berlin, Heidelberg, 2010. Springer-Verlag.