Static analysis usage for customizable semantic checks of C and C++ programming languages constraints

Valery Ignatyev
Institute for System Programming
Russian Academy of sciences
Moscow, Russia
valery.ignatyev@ispras.ru

***Abstract*—We propose the formal model of programming language constraints, which allows specifying stylistic, syntax and contextual rules. We also give the classification of those constraints. We describe the developed program model and the set of static analysis algorithms for the analyzer subsystem that implements automatic constraints checking and describe the implementation of the proposed formalizations in the Clang open source compiler.**

Keywords—static analysis; coding standards; LLVM; Clang.

I. INTRODUCTION

The C and C++ programming languages are occupying the first and fourth places in the TIOBE index [1]. Some features of these languages (like implicit type casting, direct work with memory and pointers, undefined operations) often lead to serious errors. One way of solving such problems is to impose language constraints, which can be automatically checked e.g. during compilation. Limiting the set of such restrictions allows creating lightweight analyzer with a small working time.

The set of constraints may be project specific or may contain the composition of rules from the common collections, such as MISRA [2], JSF [3], HICPP [4]. The tool for automatic checking of these rules allows simplifying development of cross platform applications via type size and addresses arithmetic control, and also improves the software security. The research of static analysis tools is performed in scientific organizations and research centers of industrial companies. Despite this there is no common formal language or model of constraints yet and the constraints classification doesn't exist.

II. FORMALIZATION AND CLASSIFICATION

A. Formal model

In papers dealing with static analysis methods, different constraint representations are used, such as Prolog [5] or a specialized declarative language [6]. There is no common model to represent it yet, so we propose to use first-order predicate calculus. The term *programming language constraint* or *rule* is defined as a predicate on the program model. Because the majority of rules are language-specific it's convenient to use compiler oriented formalizations. Constraints are defined on the program model which despite of the common representations used in compilers should contain even excess data, such as comments or white spaces. Therefore the most abstract representation is chosen. *Annotated abstract semantic graph* (AASG) is a data structure based on abstract semantic graph, each node of which is annotated and each edge is marked. Node annotation is represented with the set of attribute pairs <*name*, *value*>. AASG is based on the AST and is complemented with node attributes and edges, for example from a variable usage to its definition. Each AASG node has a semantic type, which determines the set of possible attributes. The set of all node types is limited.

The *memory area* concept – the Cartesian product of memory class and its size in bits – is used to formalize memory operations. The total of 10 memory classes are defined that are divided into 4 subclasses: (1) stack: automatic variables, function parameters, allocated with `alloca`; (2) code: function pointers, labels; (3) data: global system, global static and global memory; (4) dynamic: allocated with `malloc` and `new`. Measuring memory area size in bits allows supporting bitwise memory operations and structures with bit fields. The *memory subarea* is the Cartesian product of base memory area and its bit size; this concept allows supporting array elements and structures. The program memory state is represented as a Cartesian product of a statement identifier, a function context, an environment (variable-address mapping) and an address-value mapping. Context is ignored to speed up the analysis. Let C be a constant, V be a symbolic variable or a constant identifier, F be a symbolic identifier of a class or a structure field, then functionals α and λ defining the environment and memory area state are defined as follows: $Env = \{\alpha : Var \to A;\ \alpha^f : A_{base} \times Field \to A_{\subset};\ \alpha^e : A_{base} \times N_0 \to A_{\subset}\}$, $Mem = \{\lambda : A \to \{0,1\}^*;\ \lambda' : A \to A\}$, where A, $A_{\subset}$ is the memory area and the subarea, N_0 is the set of natural numbers and zero, $\{0,1\}^*$ – the sequence of bits in memory, which could be interpreted as a number, depending on type. The C and C++ *expression* is formalized as follows: $E := C \mid V \mid E \otimes E \mid *E \mid \&E \mid \oslash E \mid (E) \mid E \to F \mid E.F \mid E[E]$, where $\otimes := + \mid - \mid * \mid / \mid \% \mid \& \mid \mid \mid \hat{} \mid \&\& \mid \mid \mid \ll \mid \gg$ – binary operations, $\oslash := - \mid ! \mid \sim$ – unary operations. Each expression has a type corresponding to the programming language types. Each expression can have l-value (*l(E)*) and r-value (*r(E)*), which are calculated using defined rules. For example, $l(C) = \emptyset$, $r(C) = C$, $l(V) = A^v$, $r(V) = \lambda(A^V)$. The building of memory state in the analyzer consists of control flow graph traversal

978-0-7695-5194-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSTW.2014.59

and calculation of environment and memory area-value mapping during every expression subtree traversal.

B. Classification

The proposed constraints formalization allows unambiguously assigning classes to rules. Each rule uses the set of AASG nodes and its attributes, assuming its existence in the program model. However, nodes and attributes are calculated incrementally. The program analysis in a compiler is usually divided into 5 stages: lexical, syntax, semantic, optimization related and intermodule link-time analysis; likewise the set of classes is defined and rules are assigned to the classes. The proposed classification is unambiguous and describes the complexity of rule – the amount of resources, which is necessary to check the rule. The result is the set of classes: (1) *lexical* – source code formatting, comments usage, preprocessor directives and macros rules; (2) *syntax* – naming conventions, rules of using functions from standard library, format string checks; (3) *contextual* – situational constraints, depending on states of several object in program model; (4) contextual that requires additional analysis – which uses memory model, loop analysis, constant propagation and value ranges; (5) *intermodule* static rules. The last one contains rules, checked during link time, which require data from more than one compilation unit. The source data consists of specially stored and exported to database information that is collected during previous analysis phases. This class includes exceptions analysis, race conditions, whole-program tainted data analysis.

III. Developed algorithms

It's necessary to avoid symbolic interpretation and calculate only necessary data to speed up the analysis. Therefore the minimal set of commonly used attributes is calculated in advance. The constraint checking does not modify AASG and thus processing separate constraints can be easily parallelized. The intermodule analysis is executed during link time.

The expression or function *side effect* usually means the change of nonlocal data. We assume that the two expressions have *mutual side effects* if the result of their calculation may depend on the order of execution. These concepts are used in many rules, so it's necessary to develop a fast algorithm of side effects check. To do this it's enough to calculate the following AASG node attributes: (1) the set of callee functions; (2) the set of modifiable memory areas (already calculated during memory area analysis); (3) the set of thrown exceptions and (4) the set of used memory areas, to which operator lambda was applied. Thus, most of these attributes are automatically calculated during the memory model construction.

Interprocedural exception analysis algorithm checks the correspondence between thrown and handled exceptions. It is based on information stored in the annotations of statements, basic blocks and functions. The algorithm performs control flow graph depth traversal collecting and exporting to the external database the information about thrown and handled exceptions. All exported data is analyzed and error messages are generated at link time.

IV. Implementation

To implement the analyzer the LLVM/Clang [7] open source compiler infrastructure is used. The developed system consists of three main components: the subsystem defining the rules, the information collector, and the rules checking scheduler. The developed analyzer supports different build systems to handle only files compiled into the target program. The constraints and other infrastructure are written in C++, which is the Clang implementation language. The analysis is performed incrementally, as the following stages implement more complex and slower constraints. Rules are ordered automatically via the rule scheduler depending on required model nodes and attributes.

Lexical, syntax and semantic stages do not produce false positives at all. The last two stages can generate false positives for several rules, because of additional analysis algorithms drawbacks. The analyzer allows checking more than 50 different rules. Measuring the performance of the analyzer showed an average slowdown of 22% compared to the project build with disabled analyzer; this is fast enough so that the analyzer can be used at each compilation and to detect errors at the earliest possible stage.

V. Conclusion

The proposed formalization allows unambiguous constraints representation that is used for defining rules in the analyzer. The developed classification allows checker analyzing rules in the close to optimal order providing the first error messages during the analysis of more complex rules. About 50 well-known rules were implemented at the moment and the set of supported constraints will be expanded in future. The developed analyzer is used as a part of the Svace static analyzer [8] that is being deployed within the industrial customers of ISP RAS. Future research plans include supporting of other programming languages, such as Java, improving static analysis algorithms for the fast analysis with better quality, and finalizing the dissertation thesis text.

References

[1] TIOBE Programming Community Index for January 2014, www.tiobe.com

[2] Motor Industry Research Association, "MISRA-C:2004 - Guidelines for the use of the C language in critical systems", October 2004

[3] Lockheed Martin, "Joint Strike Fighter Air Vehicle C++ Coding Standards For The System Development And Demonstration Program", Document Number 2RDU00001 Rev C, December 2005

[4] Programming Research, "High Integrity C++ Coding Standard Manual - Version 3.2", http://www.codingstandard.com, 2008

[5] Marpons, Marino, Carro, Herranz, Moreno-Navarro and Polo "A Coding Rule Conformance Checker Integrated into GCC", Electronic Notes in Theoretical Computer Science, 2007.

[6] T. Matsumura, A. Monden, K. Matsumoto "The Detection of Faulty Code Violating Implicit Coding Rules", International Symposium on Empirical Software Engineering (ISESE'02), 2002, pp. 173-192

[7] Clang: a C/C++ language family frontend for LLVM. http://clang.llvm.org/.

[8] V. Nesov "Automatically Finding Bugs in Open Source Programs", Proc. of the Third Int. Workshop on Foundations and Techniques for Open Source Software Certification, 2009, vol. 20, pp 19-29