# C Code Verification based on the Extended Labeled Transition System Model

Dexi Wang, Chao Zhang, Guang Chen, Ming Gu, and Jiaguang Sun

School of Software, TNLIST, Tsinghua University, China
{dx-wang12,zhang-chao13,chenguan14}@mails.tsinghua.edu.cn

**Abstract.** The C programming language is widely used in safety-critical software systems. With its large appliance and increasing complexity, the need of ensuring the correctness of C codes emerged. This paper presents **Ceagle** , a fully automated program verifier for finding assertion violations in C programs. It is decent in both accuracy and efficiency by using a semantically equivalent program model language that is specifically designed for C program, together with various optimizations that make the satisfiability checking faster and memory-friendly. More specifically, **Ceagle** uses LLVM clang as front-end parser, an extended labeled transition system as program model, and Z3 SMT solver as the back-end satisfiability checker. **Ceagle** is designed to be fully automatic and requires no user interaction as long as the assertions are provided. For evaluation, we compare **Ceagle** with existing C program verifiers based on open benchmarks. **Ceagle** outperforms others in terms of accuracy, and time and memory consumption.

## 1 Introduction

Many safety critical software systems use C programming language, and the verification of safety properties of C programs are widely adopted to ensure the safety of the whole software system [3,5,12,13]. With the increasing complexity of system applications, the size of the C code is larger and the structure of the code is more complex, which bring new challenges for the traditional verification tools in three aspects: time efficiency, memory usage, and verification accuracy. A complex C program and corresponding safety property may easily lead to state space explosion, and the memory and time for verification may be infinite for traditional tools.

In fact, an ideal verification tool is supposed to be fast, memory friendly and accurate, but it is not easy to achieve all of them in real cases. For example, a verification tool that is very accurate may suffer from low performance, and needs a long time and huge memory to get a relative complete and sound result.

Except for the performance and accuracy, the usability of the verification tool is another important issue. Users prefer verification tools that require less user interaction than those require manual mathematical proof injection or even sophisticated formal verification knowledge background.

In this paper, we propose a novel verification tool **Ceagle** to get a better balance between the performance and accuracy, and needs less interaction from users. It is comparatively accurate and efficient by using a semantically equivalent program model language that is specifically designed for C programs, together with various optimizations that make the satisfiability checking faster. The program model language extends the original labeled transition system (LTS) with more variable types, C style statements,

and built-in functions. The extended labeled transition system (ELTS) inherits the inner property of traditional LTS that is suitable for the implementation of various model checking algorithms. Based on the ELTS, several optimizations are performed during C programs parsing, ELTS construction and ELTS analysis to make Ceagle faster and more memory friendly than other C verification tools.

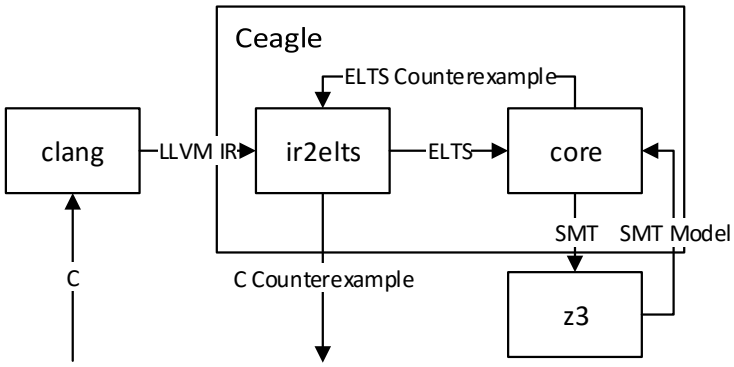


**Fig. 1.** Architecture of **Ceagle** , including the file flow among each component.

The overall structure of **Ceagle** is shown in Fig.1. **Ceagle** contains four components: *clang*, *ir2elts*, *core*, and *z3*. The C program is firstly parsed by LLVM *clang*, yielding the LLVM IR program. Then, the component *ir2elts* translates the generated LLVM IR program to the program model ELTS. Finally, the component *core* constructs SMT constraints from ELTS, and inputs them into the component *z3* for verification. If the safety property is not satisfied, the C counterexample will be generated and presented automatically by the implemented trace-back engine. During the whole procedure, the program intermediate representation LLVM IR and program model ELTS play important roles in source-level and model-level optimizations, respectively.

All model translation and construction are accomplished automatically, the user just needs to input the C program and corresponding safety property assertions. It is easy to use and does not need complex interactions or formal verification knowledge background. For evaluation, we compare **Ceagle** with existing C program verifiers based on the benchmarks, and **Ceagle** uses about 20% memory and time of CBMC to acquire almost the same accuracy, and improves the accuracy of CPAchecker for 25%.

## 2 Background

An intermediate program model ELTS is designed to represent the C program for optimized verification. Formally, an ELTS model $S$ is a five tuple $(L, l_0, V, V_0, T)$ where:

- $L$ is a finite non-empty set of locations;
- $l_0$ is the location of initial basic block;
- $V$ is a finite non-empty set of variables ranging over $\mathbb{V}$, where $\mathbb{V}$ is the (possible infinite) set of values of $V$;
- $V_0$ is the set of initial values of $V$;
- $T \subseteq L \times \mathbb{V} \times \mathbb{V} \times L$ is a finite non-empty set of transitions, a transition $t = (l_i, \mathbb{V}_i, \mathbb{V}_j, l_j) \in T$ can also be denoted as $(l_i, \mathbb{V}_i) \rightarrow (l_j, \mathbb{V}_j)$, $l_i$ and $l_j$ are called $from$ and $to$ locations, respectively.

ELTS is extended from the traditional labeled transition system to support all kinds of operations and types of C programs. For example, the built-in floating point functions such as *__fpclassifyf* and *__isnanf* contained in *math.h* are reserved in the element *T* of ELTS and will be translated to corresponding SMT constraints later during satisfiability checking. Details of ELTS can be referred to the website[1].

## 3 Related Work

During the last decades, there are approaches of assuring quality of programs by modeling and formal verification [10] [9] [7] [8]. Several verifiers for C programs are also designed to ensure the the correctness of the software such as CMBC [5], CPAchecker [3], LLBMC [12], and SMACK [13]. Among them, CMBC and CPAchecker have been widely used and accepted in both academic and industry sides. CBMC applies bounded model checking to C programs. It verifies the absence of violated assertions on a transformed GOTO program under a given loop unwinding bound, and uses SAT and SMT solver as the constraint solving back end. CPAchecker is guided by the concept of configurable software verification and aims at the easy integration of new verification components. It supports dynamic precision adjustment during analysis [2]. CPAchecker uses CDT from Eclipse as C parser and multiple SMT solvers as the back end. **Ceagle** adopt similar structure with them, but before the integration of back-end SMT, we do some optimization on the LLVM IR and use a program model ELTS which is designed to be easier for model checking, and more modular and easier to be optimized for a more efficient verification.

## 4 Design of Ceagle

As presented in Fig. 1, **Ceagle** contains four components: *clang*, *ir2elts*, *core*, and *z3*, where *clang* [11] and *z3* [6] are obtained from existing projects for C program preparser and back-end constraint solver respectively. *ir2elts* and *core* are developed for program model ELTS construction, constraint formalization and optimization.

### 4.1 Kernel Components

***LLVM clang :*** This component is used to translate C to LLVM IR. A program is firstly parsed by LLVM *clang*, yielding the LLVM IR program with debug information, the debug information will be used in C counterexample generation.

***Translator ir2elts :*** This component translates LLVM IR program to the ELTS model. Each basic block in LLVM IR program is translated into the corresponding transition in an ELTS model with four steps: (i) the name of $from$ ($to$) location in ELTS model is the label value of the current (successor) basic block in LLVM IR program; (*ii*) the guard condition of the transition in ELTS model is translated from the goto-condition of the current basic block in LLVM IR program; (*iii*) the body of the current basic block is copied as-is to preserve full LLVM IR semantics, except that LLVM IR instructions are translated into ELTS statements; (*iv*) assertion calls in LLVM IR program are translated into guarded transitions to error locations.

[1] `http://sts.thss.tsinghua.edu.cn/ceagle`

***Formalizer core :*** This component constructs SMT constraints based on the ELTS model and assertion. It performs reachability analysis of the translated error locations via a depth-first search (DFS) on the ELTS model. Once DFS reaches an error location, it constructs an SMT trace constraint containing those transitions from the initial location to current error label. It returns "TRUE" if error location is not reachable in ELTS, otherwise, it returns "FALSE" along with an ELTS counterexample, which is an ELTS trace that starts from the initial location to error location. Another function of this component is to convert the ELTS counterexample back to the C counterexample. The reverse mapping information from ELTS model to LLVM IR program is preserved by *ir2elts* when it translates LLVM IR program to ELTS model, and the reverse mapping information from LLVM IR program to C source code is preserved by *clang* via debug information stored in LLVM IR program.

***Solver z3 :*** This component checks the satisfiability of the formalized constraint, and infers violations of assertions by the following rules: (*i*) if the result is *sat* together with an SMT model, it means there exists an assignment of variables that can make the program executes to error label, thus proves an assertion violation; (*ii*) if the result is *unsat*, it means the current trace is safe, the DFS continues; (*iii*) if all traces are checked to be *unsat*, the program is proved to be safe.

### 4.2 Verification Optimizations

There are two main reasons that bring up the state space explosion problem and prevent a verifier from analyzing the C program efficiently: structural complexity and a large number of variables. We aim at reducing both of them through LLVM IR program, ELTS model, and formalization optimizations.

***LLVM IR program optimization :*** We observe that there are basic blocks in the LLVM IR that are not reachable from the program entry, which can be removed when being translated to ELTS. There are also set of basic blocks that have no conditional jumps and can be merged into one large basic block, thus helps reducing verification state space. On the other hand, an LLVM IR program is constructed in static single assignment (SSA) form [11] which introduces too many temporary variables, while an ELTS model uses C-like grammar for its transitions, which are not in SSA form. This means temporary variables introduced by SSA can be trimmed away when translated to ELTS, and multiple lines of LLVM IR instructions can be merged into one single line of C-like ELTS statement. These optimizations are carried out during the IR to ELTS model transformation.

***ELTS model optimization :*** In the ELTS model level, a depth-first search (DFS) is carried out to help to generate SMT constraints of all paths. Optimizations on ELTS model are performed to reduce the search space as follows. First, more variables are trimmed away through methods of constant propagation, temporary variables are eliminated, and locations that are not reachable to error labels are deleted, thus reduces nodes the DFS has to visit. Second, we perform optimization during the DFS search, where variables from different transitions are annotated with a sequence number according to the index of the transition in the ELTS trace, which makes it easier to construct counterexamples after satisfiability checking.

***Formalization optimization :*** A program usually contains multiple kinds of types, such as boolean, integer, and floating point, which brings several challenges to construct the

constraints in a precise and efficient way. Traditional methods solve mixed-type expressions and type conversion problems by using precise memory model [14] or explicit-state model checking [4], which is time-consuming. For optimization purpose, we implemented a lazy type conversion mechanism to infer types of expressions.

The expression is consists of variables, constants, and operators, and we encode an expression as a directed acyclic graph, in which non-leaf nodes are operators, their children are their operands, and the leaves are labeled with variables or constants. For every expression $e$, we record two types: "*intrinsic*" type $ty^i(e)$ and "*to-be*" type $ty^{tobe}(e)$. The *intrinsic* type is the expression evaluated to be, and the "*to-be*" type is the expression that should be converted to be when the expression is participating in parental expressions. The "*to-be*" type is inferred lazily, and integers and floating points are evaluated to be bit-vector "*to-be*" type when the operator is bit-precise. In this way, variables that are not necessarily to be analyzed bit-precisely can keep being arithmetic, thus reduces constraint satisfiability overhead of the back end SMT solver.

## 5 Experiment evaluations

***Experiment setup :*** We compared **Ceagle** with the most widely used and recently developed C verifiers CBMC 5.2 and CPAchecker 1.4, in terms of the accuracy, time and memory consumption. They are tested on the set of benchmarks from the software verification competition held at TACAS [1]. These are widespread benchmarks offering a good coverage of the core features of the C programming language, and many state-of-the-art analysis tools including CBMC and CPAchecker have been trained on them.

More specifically, we conducted the experiments on the 81 files in the *Floats* benchmark set, with a total of approx. 5K lines of code. The experiments are performed on a machine with a 3.4 GHz 64-bit Quad Core CPU (Intel i7-4770) processor and 32GB of memory, running an Ubuntu 14.04 with a 64-bit Linux kernel 4.2.0. We set a 15GB memory limit and a 900s timeout for the analysis of each subject.

***Verification result :*** The experiments for the three tools are summarized in Table 1. In the results table, correct negative means the file is true (i.e., negative, assertions are not violated) and the tool reported true, correct positive means the file is false (i.e., positive, assertions are violated) and the tool reported false, incorrect negative means the file is false but the tool reported true (usually means an unsound tool), incorrect positive means the file is true but the tool reported false (i.e., a false alarm, usually means an incomplete tool). **Ceagle** reported 77 correct and 4 unknown results for total 81 benchmarks, CBMC reported 78 correct and 3 unknown, and CPAchecker reported 56 correct, 1 incorrect, 6 unknown, and 18 timeout results. The correctness rate is computed by the division of the number of correct files and the number of total files (81), so the three tools have correctness rates of 95.06%, 96.30%, and 69.15%, respectively. ***Ceagle** uses about 10%-20% memory and time consumption of CBMC and CPAchecker, but acquires almost the same accuracy (just one more unknown file) of CBMC and improves the accuracy of CPAchecker for 25%.* It is reasonable to draw the conclusion that **Ceagle** is efficient in both time and memory usage while maintaining a high correctness rate. More details about efficiency are analyzed and illustrated as follows.

We presented both the time and memory consumption in the third and fourth row of Table 1. **Ceagle** is about 5.5X faster than CBMC and 5X faster than CPAchecker on arbitrary files. Note that **Ceagle** is a little bit slower than CPAchecker if only counting correct results, and it would be much faster than CPAchecker when including those incorrect and timeout results. **Ceagle** uses about 80% lesser memory than CBMC and

| | Ceagle 1.0 | CBMC 5.2 | CPAchecker 1.4 |
|---|---|---|---|
| **Number of correct files** | 77 | **78** | 56 |
| Correct negatives | 59 | 56 | 35 |
| Correct positives | 18 | 22 | 21 |
| **Number of incorrect files** | **0** | **0** | 1 |
| Incorrect negatives | 0 | 0 | 0 |
| Incorrect positives | 0 | 0 | 1 |
| **Unkowns** | 4 | **3** | 6 |
| **Timouts** | **0** | **0** | 18 |
| **Correctness rate (%)** | **95.06** | **96.30** | 69.14 |
| Total time (s) | **3700** | 21000 | 19000 |
| Total time of correct files (s) | 3700 | 18000 | **2500** |
| Time per file (s) | **45.68** | 259.26 | 234.57 |
| Timer per correct file (s) | 48.05 | 230.77 | **44.64** |
| **Time ratio of per file (%)** | 100.00 | 567.56 | 513.51 |
| **Time ratio of per correct file (%)** | 100.00 | 480.27 | 92.90 |
| Total memory (MB) | **2900** | 15000 | 31000 |
| Total memory of correct files (MB) | **2900** | 13000 | 17000 |
| Memory per file (MB) | **35.80** | 185.19 | 382.72 |
| Memory per correct file (MB) | **37.66** | 166.67 | 303.57 |
| **Memory ratio of per file (%)** | 100.00 | 517.29 | 10069.05 |
| **Memory ratio of per correct file (%)** | 100.00 | 442.57 | 806.08 |

**Table 1.** Comparison results of those tools in accuracy, time and memory consumption.

90% lesser memory than CPAchecker on arbitrary files. This is because the optimizations at different levels (LLVM IR program, ELTS model, and constraint formalization) help to reduce much verification overhead, thus reduces time and memory usage.

For the unknown and timeout result files, **Ceagle** output 4 unknown due to its lacking full support of arrays and some internal bugs on handling high order multiplications of floating points. CBMC output 3 unknown and crashed without any prompts. CPAchecker reported 6 unknown and 18 timeouts. The 6 unknown files are because of incomplete analysis. The 18 timeout files are all true files, and it is because when the *value analysis* algorithm of CPAchecker found no counterexample, the *predicate analysis* algorithm would be used, which consumed a lot of time but still could not prove true for these files.

## 6 Conclusion and future work

In this paper, we presented the C program verifier **Ceagle** to check assertion violations of C programs. It uses LLVM clang as front-end parser, an extended labeled transition system as program model, and Z3 SMT solver as the back-end satisfiability checker. Several optimization techniques during the three steps are designed and implemented to reduce the memory and time consumption. Except for those experiments on benchmarks, we have applied **Ceagle** to verify the core optimization algorithm which has 17k LOC of C code of the real train energy efficiency control software for smart railway transportation. Several safety assertions are inserted in the code to check its safety, and an over speed bug in the code is detected. Our future work mainly includes extending the scalability of **Ceagle** to support more advanced features such as struct and dynamic memory allocation of C programming language.

## References


1. D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 887–904. Springer, 2016.
2. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 29–38. IEEE, 2008.
3. D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.
4. D. Beyer and S. Löwe. Explicit-state software model checking based on cegar and interpolation. In *Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013.
5. E. Clarke, D. Kroening, and F. Lerda. *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*, chapter A Tool for Checking ANSI-C Programs, pages 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
6. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
7. Y. Jiang, Z. Li, and etc. Design and optimization of multiclocked embedded systems using formal techniques. *IEEE Transactions on Industrial Electronics*, 62(2):1270–1278, 2015.
8. Y. Jiang, H. Liu, and etc. Design of mixed synchronous/asynchronous systems with multiple clocks. *IEEE Transaction on Parallel and Distributed Systems*, pages 2220–2232, 2015.
9. Y. Jiang, H. Song, and etc. Data-centered runtime verification of wireless medical cyber-physical system. *IEEE Transactions on Industry Informatics*, 2016.
10. Y. Jiang, H. Zhang, and etc. Tsmart-galsblock: a toolkit for modeling, validation, and synthesis of multi-clocked embedded systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 711–714. ACM, 2014.
11. C. Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.
12. F. Merz, S. Falke, and C. Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *Verified Software: Theories, Tools, Experiments*, pages 146–161. Springer, 2012.
13. Z. Rakamarić and M. Emmi. Smack: Decoupling source language details from verifier implementations. In *Computer Aided Verification*, pages 106–113. Springer, 2014.
14. C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In *Proceedings of the 5th international conference on Systems software verification*, pages 7–7. USENIX Association, 2010.