Boosting Software Verification with Compiler Optimizations

Gyula Sallai* and Tamás Tóth†
Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
Fault Tolerant Systems Research Group
Email: *salla@sch.bme.hu, †totht@mit.bme.hu

***Abstract*—Unlike testing, formal verification can not only prove the presence of errors, but their absence as well, thus making it suitable for verifying safety-critical systems. Formal verification may be performed by transforming the already implemented source code to a formal model and querying the resulting model on reachability of an erroneous state. Sadly, transformations from source code to a formal model often yield large and complex models, which may result in extremely high computational effort for a verifier algorithm. This paper describes a workflow that provides formal verification for C programs, aided by optimization techniques usually used in compiler design in order to reduce the size and complexity of a program and thus improve the performance of the verifier.**

I. Introduction

As our reliance upon safety-critical embedded software systems grows, so does our need for the ability to prove their fault-free behavior. Formal verification techniques offer reliable proofs of a system's correctness. These algorithms operate on formal models which describe the semantic behavior of the system under verification and are able to answer queries on its properties. However, model-driven design can be rather difficult and the financial and time constraints of a project often do not make it a viable choice for development.

Many projects start right at the implementation phase without sufficient planning and modeling. In the domain of embedded systems, implementation is usually done in C. Although there are many tools that can be used to generate C code from a formal model (model-to-source), the reverse transformation (source-to-model) is far less supported.

Another difficulty with this process is the size of the state space of the model generated from the source code. As most verification algorithms have a rather demanding computational complexity (usually operating in exponential time and beyond), the resulting model may not admit efficient verification. A way to resolve this issue is to reduce the size of the generated model during source-to-model transformation.

The project presented in this paper proposes a *transformation workflow* from C programs to a formal model, known as control flow automaton. The workflow enhances this transformation procedure by applying some common *optimization transformations* used in compiler design [1]. Their application results in a simpler model, which is then split into several smaller, more easily verifiable chunks using the so-called *program slicing* technique [2]. This allows the verification algorithm to handle multiple small problems instead of a single large one. At the the end of the workflow, the resulting simplified slices are verified using *bounded model checking* [3] and *k-induction* [4].

Measurements show that the applied transformations reduced the models' size considerably, making this technique a promising choice for efficient validation. Benchmarks on the execution time of the verifier algorithm suggest that breaking up a larger program into several smaller slices may also speed up the verification process.

†This work was partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

II. Background and Notations

There are several program representations with formal semantics suitable for verification. In this paper we shall focus on the one called *control flow automaton* (CFA) [5]. A CFA is a 4-tuple (L, E, ℓ_0, ℓ_e), where

- $L = \{\ell_0, \ell_1, \dots, \ell_n\}$ is a set of locations representing program counter values,
- $E \subseteq L \times Ops \times L$ is a set of edges, representing possible control flow steps labeled with the operations performed when a particular path is taken,
- $\ell_0 \in L$ is the distinguished entry location, and
- $\ell_e \in L$ is the special error location.

During verification we will attempt to prove that there is no feasible execution path which may reach ℓ_e, thus proving that the input program is not faulty. Let π be a path in a CFA (L, E, ℓ_0, ℓ_e). We say that π is an *error path* iff its last location is ℓ_e. π is an *initial path* iff its first location is ℓ_0. The path π in a CFA is an *initial error path* iff its both an initial path and an error path.

The verification algorithms used in our work are *bounded model checking* and *k-induction*. A bounded model checker [3] (BMC) searchers for initial error paths with the length of k (the *bound*) and reduces them to SMT formulas. If the resulting formula is satisfiable, then its solution will serve as a counterexample to correctness. If no satisfiable formula was found for a length of k, then the algorithm increases the bound to $k + 1$. It repeats this process until it finds a counterexample or reaches a given maximum bound. Bounded model checking is not complete, and can only be used for finding counterexamples in erroneous programs, as the BMC algorithm always runs into timeout for safe programs.

For proving safety, we can use k-induction [4]. A k-induction model checker applies inductive reasoning on the length of program paths. For a given k, k-induction first proves that all paths from ℓ_0 with the length less than k are safe, using bounded model checking. If the BMC algorithm finds no counterexamples then the algorithm performs an induction step, attempting to prove that a safe path with the length of $k-1$ can only be extended to a safe path with the length of k. This is done by searching for error paths with the length of k and proving that their respective SMT formula is unsatisfiable. If all error paths with the length of k are unsatisfiable then all initial error paths will be unsatisfiable, thus the safety of the system is proved. If a feasible error path exists then it is a counterexample to the induction and the safety of the program cannot be proved of refuted with the bound k.

In order to reduce the resulting model's size, we shall use *optimization transformations* usually known from compiler theory. Compiler optimizations transform an input program into another semantically equivalent program while attempting to reduce its execution time, size, power consumption, etc [1]. In our work we used these transformations to reduce the resulting model's size and complexity. Many of these optimization algorithms are present in most modern compilers. The project presented in this paper focuses on the following algorithms:

- constant folding,
- constant propagation,
- dead branch elimination,
- function inlining.

Constant folding evaluates expressions having a constant argument at compile-time. Constant propagation substitutes constants in place of variables with a value known at compile-time. Both algorithms operate on local and global constants. In many cases, these two algorithms are able to replace one or more branching criteria with the boolean literals **true** or **false**. Dead branch elimination examines these branch decisions and deletes inviable execution paths (e.g. the **true** path of a branch decision always evaluating to **false**). Function inlining is the procedure of replacing a function call with the callee's body. In this work we shall use function inlining to support simple inter-procedural analysis, as an inlined function offers more information of its behavior than a mere function definition.

This work also makes use of an efficient and precise program size reduction technique known as *program slicing*. Weiser [2] suggested that programmers, while debugging a complex program, often dispose code pieces irrelevant to the problem being debugged. This means that programmers usually mentally extract a subset from the entire program relevant to some criteria. He called these subsets *program slices*. Attempting to formalize this practice, Weiser defined a program slice P' as an *executable subset* of a program P, which provides the same output and assigns the same values to a set of variables V as P at some given statement S. This statement S and the variable set V is often put together into a pair which will serve as the *slicing criterion*. By using slicing with multiple criteria, it is possible divide a larger program into several smaller executable slices.

III. Contribution

The project presented in this paper implements a verification compiler, that is a compiler built to support verification. This is done by using a complex workflow which transforms C source code to control flow automata, applying optimization transformations and program slicing during the process. The resulting model(s) can then be verified using an arbitrary verification algorithm. An overview of the workflow can be seen in Figure 1.

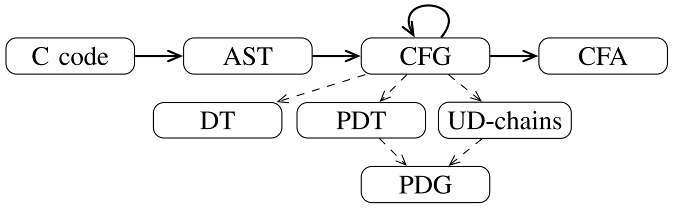

Fig. 1. Transformation workflow.

As an input, the compiler takes a C source code file, which is then parsed and transformed into an *abstract syntax tree* (AST), representing the syntactic structure of the program. This AST is then transformed into a *control flow graph* (CFG), representing the instructions and control flow paths of the program. Optimization algorithms and program slicing are performed on the CFG, resulting in multiple smaller CFG slices of the program. These slices then are transformed into control flow automata. Currently the slicer criteria are the assertion instructions in the control flow graph (which are calls to the `assert` function in C), meaning that each assertion gets its own CFA slice. In the resulting CFA, the error location represents a failing assertion.

Several helper structures are required for these transformations, such as *call graphs* (for function inlining), *use-definition chains* for data dependency information, *dominator trees* (DT) and *post-dominator trees* (PDT) for control structure recognition [1]. The program slicing algorithm requires the construction of a *program dependence graph* (PDG), which is a program representation that explicitly shows data and control dependency relations between two nodes in a control flow graph. The control dependencies show if a branch decision in a node affects whether another instruction gets executed or not. Data dependencies tell which computations must be done in order to have all required arguments of an instruction. Slicing is done by finding the criteria instruction S in the PDG and finding all instructions which S (transitively) depends on [6].

IV. Implementation and Evaluation

The implemented system has three main components: the parser, the optimizer and the verifier. The parser component handles C source parsing and the control flow graph construction. The optimizer module performs the optimization transformations and program slicing and is also responsible for building the control flow automata from the CFG slices. The verifier component (implemented as a bounded model checker with k-induction) performs the verification on its input model.

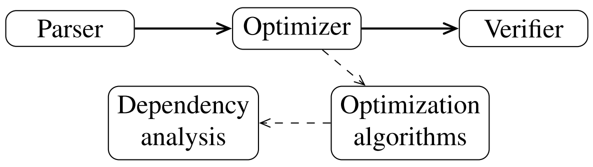

Fig. 2. Architecture of the implemented program.

All components are implemented in Java 8, with dependencies on certain Eclipse[1] libraries. The program also makes use of the theta formal verification framework, developed at the Fault Tolerant Systems Research Group of Budapest University of Technology and Economics. It defines several formal tools (mathematical languages, formal models) and algorithms. It also provides a set of utilities for convenience, such as expression representations and interfaces to SAT/SMT solvers, which are used in the project's implementation. The work discussed here extends this framework with an interface and toolset for C code verification.

The parser module utilizes the parsing library of the Eclipse C/C++ Development Tools plug-in (CDT). The CDT library performs lexing and parsing and returns an abstract syntax tree, which is then transformed into a control flow graph. Currently only a small subset of the C language is supported. The current implementation only allows the usage of control structures (such as **if-then-else**, **do-while**, **switch**, **while-do**, **break**, **continue**, **goto**) and non-recursive functions. Types are only restricted to integers and booleans. Arrays and pointers are not supported at the moment.

The optimizer module handles optimization transformations and program slicing. The implemented transformation algorithms are constant folding, constant propagation, dead branch elimination, function inlining and program slicing. After finishing with the optimization and transformation passes, the optimizer generates a list of control flow automata from each extracted slice. These smaller slices then later will be used as the verifier's input.

Currently the verifier is implemented as a simple bounded model checker extended with a k-induction algorithm. The verifier operates on a collection of control flow automata, with each automaton being a slice extracted from the input program. If a CFA was deemed faulty, then the whole program is reported as erroneous. Currently the verifier may report one of the following statuses: `FAILED` for erroneous programs, `PASSED` for correct programs and `TIMEOUT` if it was not able to produce an answer in a given time limit.

To evaluate the effects of the optimizations mentioned previously, we shall use two types of measurements: the size of the control flow automata used as the verifier input and the results of a benchmarking session on the verifier execution time. The size of an automaton is currently measured by two factors: the number of its locations and edges. The performance benchmarking was performed by measuring the execution time of the verifier on every input CFA. Due to the slicing operation, a single input model may get split into several smaller slices, which then can be verified independently.

The verification task sets are divided into three categories, two of them are taken from the annual *Competition on Software Verification* (SV-COMP) [7]. The first task set, **trivial**, contains trivially verifiable tasks, such as primitive locking mechanisms and greatest common divisor algorithms. The task sets used from the SV-COMP repertoire are the ones called **locks** and **eca**. The **locks** category consists of programs describing locking mechanisms with integer variables and simple **if-then-else** statements.The **eca** (short for *event-condition-action*) task set contains programs implementing event-driven reactive systems. The events are represented by nondeterministic integer variables, the conditions are simple **if-then-else** statements.

The results are shown with two different optimization levels. The first level only uses function inlining, as it is needed for verifying some interprocedural tasks. The second level utilizes all optimizing transformations presented in this paper, including function inlining and program slicing.

The measurement results for each optimization level are shown in different tables. The first column always contains the task name, while the other columns contain the measurement and benchmarking data for a given slice. The legend of column labels is shown in Table I.

TABLE I
COLUMN LABELS AND THEIR ASSOCIATED MEANINGS.

Label	Description
L	CFA location count
E	CFA edge count
R	Verification result (**F**AILED/**P**ASSED/**T**IMEOUT)
ER	Expected verification result (**F**/**P**)
T	Verification execution time (average of 10 instances, in ms)
S	Slice count (for sliced programs)
SL	Average location count (for sliced programs)
SE	Average edge count (for sliced programs)

All models were checked with the timeout of 5 minutes on a x86_64 GNU/Linux (*Arch Linux with Linux Kernel 4.7.6-1*) system with an Intel i7-3632QM 2.20 GHz processor and 16 GB RAM.

TABLE II
BENCHMARK RESULTS WITH INLINING ONLY.

Task	L	E	R	ER	T
triv-lock	8	8	F	F	5
triv-gcd0	11	11	F	F	4
triv-gcd1	9	9	F	F	18
locks05	62	86	T	P	-
locks06	53	73	T	P	-
locks10	98	138	T	P	-
locks14	136	194	F	F	33
locks15	145	207	F	F	36
eca0-label00	391	459	T	F	-
eca0-label20	391	459	T	F	-
eca0-label21	391	459	T	F	-

[1]http://www.eclipse.org/

The benchmark results without any optimization algorithms (except inlining) are summarized in Table II. As it can be seen, the erroneous tasks can usually be verified rather fast, except for the **eca** task set. This set contains models with large if-else constructs inside loops. This yields an exponential number of possible error paths. The bounded model checking algorithm is rather ineffective for such problems and thus, it cannot handle these models in a reasonable amount of time, resulting in a timeout in all cases. It is also worth noting that the k-induction algorithm could not prove the non-faulty models' correctness within the given time frame.

TABLE III
BENCHMARK RESULTS WITH FULL OPTIMIZATION.

Task	S	SL	SE	R	ER	T
triv-lock	1	8	8	F	F	6
triv-gcd0	1	11	11	F	F	4
triv-gcd1	1	9	9	F	F	20
locks05	6	18	23	P	P	9
locks06	5	17	21	P	P	9
locks10	10	22	29	P	P	14
locks14	16	33	45	F	F	25
locks15	17	33	47	F	F	27
eca0-label00	1	309	377	T	F	-
eca0-label20	1	309	377	T	F	-
eca0-label21	1	309	377	T	F	-

Benchmark results with optimization are listed in Table III, which shows the number of produced slices, their average location and edge count (rounded to the nearest integer) and also the verifier running time. As it is sufficient to find one failing assertion among all slices for reporting that the input program is faulty, the running time for erroneous programs is the time until the verifier found the first failing slice. For correct programs, the running time is equals to the sum of the running time for all slices.

Table III shows that while the optimization transformation have little to no effect on trivial programs, it reduces the size of larger programs considerably. While the non-faulty programs of the **locks** category have all ran into timeout without optimization, their verification finished almost instantly after the optimization transformations. Due to the small running time, the other running time measurement differences are within the margin of error.

The tasks of the **eca** set were also reduced considerably, location count is reduced by 21%, edge count is reduced by 17%. Sadly, the verifier algorithm was not able to cope even with the reduced programs of this set, still timing out during verification. As the verification method is completely replaceable and this bounded model checking was merely implemented for the workchains completeness, this is not a large issue. However, further investigation is required for execution time evaluation.

V. CONCLUSIONS AND FUTURE WORK

In this paper we described a transformation workflow for generating multiple smaller optimized formal models from a single C program. To achieve this, the workflow uses optimization algorithms known from compiler theory and the program slicing technique.

The resulting models are then verified using a simple bounded model checking and k-induction algorithm. The developed project was built as modular components, therefore any module can be replaced for further improvement.

The evaluation of the above methods showed that program slicing is promising technique for program size reduction especially for verification. It is also worth noting that splitting a larger problem into multiple ones may allow efficient parallelization of the verification algorithm. As the runtime evaluation proved to be difficult because of the implemented verifiers performance, further evaluation is in order with other, more effective verification algorithms.

The project has several opportunities for improvements and feature additions. Some of them are listed below.

- Extending the support for more features of the C language. Such features could be arrays, pointers, structs.
- Introducing other optimization algorithms into the workflow, such as interprocedural slicing, or more aggressive slicing methods such as value slicing [8].
- Currently the counterexample is only shown in the verified formal model. The additions of traceability information would allow showing the counterexample in the original source code.
- The LLVM compiler infrastructure framework[2] provides a language-agnostic intermediate representation (*LLVM IR*) for several programming languages. Adding support for the LLVM IR would extend the range of supported languages and would also implicitly add multiple fine-tuned optimizations into the workflow.

REFERENCES

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

[2] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. IEEE Press, 1981, pp. 439–449.

[3] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, 1999, vol. 1579, pp. 193–207.

[4] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '00. Springer-Verlag, 2000, pp. 108–125.

[5] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Formal Methods in Computer-Aided Design*. IEEE, 2009, pp. 25–32.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[7] D. Beyer, "Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016)," ser. Lecture Notes in Computer Science, M. Chechik and J.-F. Raskin, Eds. Springer Berlin Heidelberg, 2016, vol. 9636, pp. 887–904.

[8] S. Kumar, A. Sanyal, and U. P. Khedker, "Value slice: A new slicing concept for scalable property checking," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag New York, Inc., 2015, pp. 101–115.

[2]http://llvm.org/