Finding More Bugs with Software Model Checking using Delta Debugging

Oleg Petrov
Lomonosov Moscow State University
Leninskiye Gory, 1-52, Moscow, 119991, Russian Federation
Ivannikov Institute for System Programming of the Russian Academy of Sciences
Alexander Solzhenitsyn st., 25, Moscow, 109004, Russian Federation
o.petrov@ispras.ru

***Abstract*—Many verification tasks in model checking (one of the formal software verification approaches) can't be solved within bounded time requirements due to combinatorial state space explosion. In order to find a bug in the verified program in a given time, a simplified version of it can be analyzed. This paper presents DD** algorithms (based on the Delta Debugging approach) to iterate over simplified versions of the given program. These algorithms were implemented in software-verification tool CPAchecker. Our experiments showed that this technique can be used to find new bugs in real software.**

***Index Terms*—formal software verification, model checking, delta debugging, CPAchecker**

I. Introduction

A significant portion of tasks and problems today are solved with the aid of software. With the increase in the scale and complexity of tasks, the scale and complexity of the software systems that solve them increase, as does the difficulty of preventing, detecting, and eliminating errors in them.

Approaches to detecting errors in programs can be divided into three types: expertise, dynamic analysis, and static analysis. Expertise is the manual review of code (or other development artifacts) by a human with a high enough level of expertise and is not scalable. Dynamic analysis methods involve the analysis of a sufficiently long run of the software system or the analysis of test runs. It can be automated, but it can only detect bugs on paths that were included in the test suite and cannot prove program correctness.

Static analysis includes methods for analyzing the source code of a program or its binary code without running the program. Program analysis (lightweight static analysis techniques such as control flow analysis and data flow analysis) is thoroughly used in compilers [1] and can be used to detect probable defects in a short time. On the other hand, formal verification methods make it possible to reliably obtain evidence of an error (counterexample) or even prove the absence of errors (correctness of a program with respect to a given formal specification), but this may require significant computational resources or human aid.

One of the most successful tools for automatic model checking of C programs is CPAchecker[1] [2], [3]. With its help, several hundred errors were found in the code of the Linux operating system drivers [4]. The tool is actively developed[2] and wins medals in the program verification competitions SV-COMP several years in a row [5]–[7].

Although at the 2022 competition this tool received second place in the summary category Overall, it was unable to complete the verification of a considerable number of programs due to a 15-minute CPU time limit. Table I compares the CPAchecker verification tool and the winners in the corresponding competition categories in terms of the number of programs that were verified within the allotted time. Points in the competition were awarded for the correct verdict with a correctness or violation witness that was validated, but in this table, the correctness of the verdict and witness were not taken into account because they require validation by hand or by other tools due to the complexity of real software systems.

The table shows that even the winners in the respective categories failed to verify a significant portion of programs, especially in the SoftwareSystems category, which consists of complex programs that are close to the real software systems used. The obvious solution to the lack of resources for verification is to allocate more resources, but often this does not help get a verdict.

In this work, we use the approach of simplifying the verified program. This approach is known, but we have proposed an automatic approach to the systematic enumeration of simplified versions of the program. For this, an algorithm based on the Delta Debugging algorithm was proposed. The implementation manipulates (removes) function bodies from the internal representation of the program in CPAchecker, a control flow automaton. The enumeration with the help of the proposed algorithms takes a significant amount of time, and its limitations lead to the loss of up to 38% of verdicts that the baseline analysis could find, but in this way, it was still possible to find 32% *unsafes* that the baseline analysis could not find in the same time. Due to the complexity of proving the correctness of the original program on the basis of the correctness of simplified programs, the search for *safe* verdicts remains outside the scope of this work.

[1] https://cpachecker.sosy-lab.org/

[2] https://gitlab.com/sosy-lab/software/cpachecker

Table I
Programs verified, SV-COMP 2022

Category	Programs in category	Verified by CPAchecker	Winner in category	Verified by winner
ReachSafety	5400	3477 (64%)	VeriAbs	4476 (83%)
MemSafety	3321	2992 (90%)	Symbiotic	3264 (98%)
ConcurrencySafety	763	377 (49%)	Deagle	559 (74%)
NoOverflows	454	369 (81%)	CPAchecker	—//—
Termination	2293	1023 (45%)	UAutomizer	1589 (69%)
SoftwareSystems	3417	1830 (54%)	Symbiotic	1261 (37%)
FalsificationOverall[a]	13355	3726 (28%)	CPAchecker	—//—
Overall[b]	15648	10195 (65%)	Symbiotic	8962 (57%)

[a]All previous categories except Termination.
[b]All previous categories including Termination.

II. Related work

There are different techniques that can be applied in model checking in order to obtain results: a) specific to the problem of combinatorial explosion in model checking, b) general-purpose techniques for reduction of the software to be verified, and c) reuse of partial results of verification.

A. Model checking techniques

Model checking is a formal software verification technique, i.e. a program is checked against specification—some formally expressed property (often in a from of a temporal logic formula [8]). Model checker explores state space of the given program and checks seen states against the given specification simultaneously. A state of a program is values of all its variables and current control location (value of instruction pointer).

When a state violates the given specification, model checker can export a *counterexample* — a trace to this state — as a specification violation witness. This possibility of systematic search for error paths makes model checkers potent tools for bug-finding.

One of the well-known techniques to reduce generic software model is abstraction. Explicit model of a program is overapproximated by an abstract model in a way, that does not lose counterexamples. Abstraction is often paired with counterexample-guided abstraction refinement [9]. This way, model checker starts with the most abstract model; when a *spurious* counterexample is present in the abstract model, but is not feasible in the software verified, it is used to make the abstraction more precise. Abstract model is refined this way until a feasible counterexample is found or whole model is checked.

Other classic techniques include partial order reduction (taking into account that some asynchronous events simulated in a different order lead to the same state [10]), and symmetry reduction (using symmetry in systems with multiple identical components [11]), both of which are used especially for model checking of concurrent systems; symbolic model checking (using binary decision diagrams as compact encoding of state space [12]).

Bounded model checking [13] bounds depth of model exploration, and therefore either provides counterexample shorter than the imposed limit, or proves that there are no such counterexamples. This technique is throughly improved and is used in practice for bug-finding.

B. Partial verification and verification of parts

Another approach for state space reduction is to reduce the input program before modelling it. One well-known approach that can be viewed as program simplification technique is program slicing [14]: only statements that affect values of the given variables at the given instructions (*slicing criterion*) through control or data flow remain in program. This technique was evaluated with CPAchecker twice [15], [16] with mixed results, and was implemented [17] as a *configurable program analysis* inside CPAchecker (i.e. it can be used alongside other CPA to verify a given program [3]).

Usually large-scale software systems are divided into components. Software verification can benefit off this structure via interface rule, assume-guaranty reasoning, or other techniques oriented on component-based software verification [18]. Contrarily, decomposition of specification can also be useful [19].

Incremental verification [18] and extreme model checking [20] can be used with incremental software system development and extreme programming, respectively. This way software verification benefits from the fact that most part of the software system was already verified, therefore verification of the new version of the software is approachable.

Another technique that is especially useful for regression verification is precision reuse [21]. In similar fashion, the *precision* of abstract model of the software older version can be used to achieve efficient verification of the newer version.

Conditional model checking [22] proposes to export partial results of a verification run as a predicate describing safe (explored) part of the verified software and add such predicate as an input to a verification tool. *Safe* verdict is represented as *true*, and *unsafe* verdict is represented as *false*. This way different tools can exchange information.

The state-of-the-art verification tools make it possible in practice to increase the efficiency of verification by transferring information between two tools (or a tool running in different configurations). A tool and language "for the composition of cooperative approaches" have been proposed [23]. At the SV-COMP 2022 competition [6], such a tool could have taken second place in the ReachSafety, MemSafety, and Termination categories and first place in the NoOverflow category, but it did not participate in the rating because it used other participating instruments.

C. Delta Debugging

This paper proposes the automatic enumeration of simplified versions of the program being verified. This technique is closer to the verification of parts of the program. The most known approach to changing input data, program version, or other startup conditions is Delta Debugging, proposed by [24]. These algorithms iterate over subsets of a set of arbitrary homogenous atomic elements that make up the "changeable circumstances". The initial set is split into smaller parts, *deltas*, and for both deltas and their complements the interesting property can be checked. Then deltas are split into ever smaller parts, until they consist of one element.

In this paper, function bodies of an original analyzed program are considered elements, i.e., simplified versions of the same program miss some function bodies. Lines of code, blocks, and operators can also be considered as less coarse elements.

Delta Debugging distinguishes three outcomes in terms of a test run outcome. Let original full set of input elements holds some property *fail* (i.e., test run produces a failure; here, a model checker can not verify a given program in a given time). Let empty set of input elements (baseline) holds some property *pass* (i.e., test run succeeds; here, a model checker provides a *safe* or *unsafe* verdict, which is the case for an "empty" C program of `int main() { return 0; }`). These two properties must be mutually exclusive (test can not succeed and fail simultaneously). The case when neither is held is considered *unresolved* (here, an error occured in the verification tool). Seminal work proposes three algorithms based off the same approach:

- *ddmin*: minimization of fail-inducing subset;
- *ddmax*: maximization of passing subset;
- *dd*: isolation of a fail-inducing difference ("cause").

As these algorithms do not enumerate all of the subsets, the minimum (maximum) found by ddmin (ddmax) is local. The authors call it 1-minimal (1-maximal), as no element in the found subset can be removed so that *fail* holds (no element can be added so that *pass* holds). When dd finds a "cause", that means that there is some "safe" subset for which *pass* holds, but for the "safe" subset together with the "cause" the *fail* holds.

Delta Debugging improvements: The dd algorithm can work with an unstructured set of elements, whether they are commits, user actions, files, lines, HTML tags, tokens, characters. Ignoring the internal structure of the input allows the algorithm to be used in a wide range of situations, but also allows a large number of unnecessary runs due to ignoring information about internal dependencies.

A Hierarchical Delta Debugging (HDD) algorithm has been proposed that is capable of minimizing tree-structured data faster and effective than ddmin [25]. This algorithm uses ddmin to minimize each level of the input tree, starting from the root, and removing nodes with their entire subtrees. Authors applied HDD to minimize C programs in form of an abstract syntax tree.

Other improvements and applications of the DD algorithms include subtree hoisting [26] and binary reduction of dependency graphs (e.g. applicable for Java classes) [27].

III. General design

We simplify the verified program (by removing its parts) in order to find an *unsafe* that is also feasible in the original program. This means that it is necessary to 1) propose and implement an algorithm for enumerating simplified versions of the program; 2) implement a check of a counterexample found in a simplified version against the original version of the program.

Accounting for both of these problems, we need to mutate original program until am unsafe

When an *unsafe* was found on some program control flow automaton modified in this way, the resulting counterexample is checked against the restored control flow automaton. If the *unsafe* has been confirmed, the algorithm terminates, otherwise the enumeration process continues.

As a result, the following cycle was implemented inside the CPAchecker tool.

1) CPAchecker parses the program and builds its control flow automaton (CFA).
2) CPAchecker starts verification of the program with the time limit specified for one verification round.
3) If a verdict is produced, CPAchecker returns it; otherwise timeout has occured (*fail* outcome in terms of DD).[3]
4) If there is no way to mutate the CFA of the program or the time allotted for the whole process has run out, exit with the *unknown* result.
5) Otherwise, change the program CFA. DD chooses what to do based on the results of previous verification round.
6) CPAchecker starts verification with the time limit specified for one verification round.
7) If an *unsafe* verdict is produced, check the counterexample.
8) If the counterexample is confirmed against the original program, CPAchecker returns the *unsafe* verdict.
9) Otherwise, go to step 4. For DD, *unsafe* and *safe* mean *pass* outcome, and timeout means *fail*.[3]

[3]In practice, other problems can occur (such as exceptions thrown by the verification tool), but here we consider only *safe*, *unsafe*, and timeout possible for simplicity.

A. Simplification problem

The main question is how to arrange a sufficiently fast enumeration of simplified versions of the program. In the following, we are considering only removing function bodies, as it makes sense to remove coarse elements of the input program before removing more fine-grained elements like blocks and statements, and this case has been implemented and evaluated.

On the one hand, the more complex the function, the more likely it (or the code that uses it) has a bug. On the other hand, the analysis of complex functions is also resource intensive. Thus, to increase efficiency, it is necessary to separate the possibility of finding an error when calling a function from the complexity of its analysis. In addition, it is worth considering that a large number of simple functions can be worse than a few complex ones.

The complexity of a function can be estimated through the characteristics of its control flow automaton as a graph: the number of vertices, edges, cycles, its cyclomatic complexity, whether there are sink vertices in the function (the possibility of early termination of the entire program); the semantic characteristics of a function as a program: the number of variables, pointers, function calls in it and whether it calls itself, is it a pure function or does it have side effects; finally, how many times the analysis entered certain locations of the function.

The presented problem can be reformulated as a knapsack problem: it is necessary to choose as many interesting (here value is probability of an *unsafe*) functions as possible so that the analysis does not exceed resource constraints (i.e. weight is an estimate of the is complexity of a function for analysis).

In such setting, it is enough to enumerate the largest sets of functions, for which the verification completes before the allotted time limit, since smaller subsets of such a set can only miss an unsafe. Such a maximum set can be found using Delta Debugging, with timeout being the *fail* outcome, and verdicts *safe* and *unsafe* being the *pass* outcome.

Contrarily, it may be interesting to find a minimum set of functions that can be called a core of complexity, as the verification of this set ends in a timeout. As the ddmin algorithm approaches minimum, it tries some of its subsets too, including removing each function from minimum set individually.

Thus, the proposed algorithm for enumerating simplified versions is based on the previously implemented dd algorithm, which localizes the cause. Based on it, algorithms dd*min* and dd*max* were developed for searching for a suitable configuration by enumeration of minima and, accordingly, maxima.

B. Iterative algorithms DD**

The ddmin algorithm can be used to find the minimum set of functions each of which is required to reproduce the timeout. Below a dd*min algorithm is proposed for finding the minimum set of causes, since we may be interested in the structure of the minimum set of functions, i.e., which functions together form “causes”. dd*min showed speed comparable to ddmin.

To search for functions without which a timeout does not occur, the dd algorithm can be used. The first run of dd will split the set of functions into three sets: the set of removed functions, the set of “safe” functions (which the verification tool manages to analyze in the allotted time), and the isolated “cause”, i.e., the set of functions, after adding which to the set of “safe” functions a timeout reappears.

By repeating dd on the set of safe functions, we can isolate a new cause among them (and remove some of these functions, adding them to the set of removed functions). dd is repeated until the set of safe functions is empty; now we have a set of removed functions and a set of isolated causes, which makes up the minimum program that the verification tool can not verify in the allotted time.

Similarly, you can find the maximum program not with the ddmax algorithm, but by iteratively removing causes with dd*max. To do this, the cause is deleted after each run, and all the functions that were removed on this run are returned. This way a new cause can be isolated among all other functions. The process continues as long as the timeout continues to occur after the return of the removed functions. Thus, we get a set of causes that have been removed from the program, and a set of safe functions.

It is possible to construct an algorithm that enumerates the optimums based on algorithms that find a local optimum. In the following, two such algorithms, dd*min* and dd*max*, are described.

To iterate over minima, it is enough to return all removed functions and remove one of the isolated causes. If the timeout does not occur without this cause, then we return it and try to remove another one. If the timeout reoccurs, then we can find another minimum, since it will not have the cause that we removed. This way all the causes found can be removed one by one. In like fashion, it is enough to add one of the causes to the found maximum to find another maximum by isolating another cause.

Taking into account that DD’s complexity with respect to the number of analysis runs performed is linear in the number of considered elements, we obtain, in the worst case, a quadratic dependence on the number of elements. Assuming that the number of causes in the found minimum is bounded from above by some constant, we obtain a linear complexity estimate (with the indicated constant as a factor).

C. Counterexample check

CPAchecker has three implementations for checking counterexamples: using CBMC (Bounded Model Checker for C and C++ programs[4]), concrete execution, and using CPAchecker itself. In the first two cases, the found counterexample is exported as a C program. In the latter case, it is exported as a violation witness in the form of a special automaton that directs the analysis along the already found trace [28].

[4]http://www.cprover.org/cbmc/

Since translated programs or a violation witness significantly limit the number of possible execution paths of the program, their analysis is much easier than the analysis of the complete original program. Because of that, more complex analyses can be used to confirm *unsafes* found with simple analyses.

When checking a counterexample, it is necessary to correct the representation of the error trace in order to compensate for the fact that it was found on a modified program. For representation as a program, definitions of removed functions have to be added.

To check a counterexample found for a simplified version of the program, the following was implemented. The counterexample is translated into C in much the same way as for CBMC, but the definitions of the removed functions are added to the resulting text. Then re-verification is started from within CPAchecker (by default with the same configuration). Although there is now a potentially complex function, the rest of the program has been simplified to a single trace, so this check requires much less resources compared to the entire program.

IV. Evaluation

Two experiment were conducted to evaluate implemented algorithms, both compare dd*min* and dd*max* against the baseline CPAchecker analysis with the same CPU time limit. Effectiveness is evaluated as amount of found *unsafes*, efficiency is evaluated as time spent for the tasks.

A. A few programs from SV-COMP/ReachSafety

29 programs were chosen arbitrarily for the first experiment from ReachSafety category of the SV-COMP benchmark[5]. These programs are checked for reachability of specified function call (reachable call is considered a bug). 21 of the chosen programs have a bug (the call is reachable) and 8 of the programs do not have a bug (the call is not reachable). Most of the programs consist of a few functions, some have a lot of branching. For each of the chosen programs, CPAchecker did not provide a verdict due to timeout (15 minutes of CPU time).

2.5 hours of CPU time (9000 seconds) were allocated for verification of one program. The run was performed using BenchExec[6] on a machine with a 16-core 11th generation Intel Core i7-11700 processor at 2.50 GHz, with 32 GB of RAM (of which CPAchecker had allocated 10 MB on the heap and default 1 MB on the stack), and 64-bit operating system Ubuntu 20.04.6 LTS.

Baseline configuration is `-svcomp22 -benchmark` (without forced timelimit). DD** configuration run same analysis with time limit of 200 s for each verification round.

As seen in 1, baseline analysis found 6 *unsafes* (out of 21 programs with an error) and 0 *safes* (out of 8 programs without an error), while both dd*min* and dd*max* found only two *unsafes*, and one unsafe was found by all three.

[5]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[6]https://github.com/sosy-lab/benchexec

Table II
Results of Linux device drivers verification

	Baseline analysis	dd*max*	dd*min*
Total CPU time, h	493	142	240
Total wall time, h	427	131	218
Safe	90	74	75
Unsafe	62	49	77
Enumeration completed	—	130	50
Timeout	100	0	1
Out of memory	6	3	12
Outside problems	21	21	21
Recursion in module	5	15	13
Other exceptions	0	7	46

In total, baseline analysis took 161 hours of CPU time (44.8 hours of wall time), dd*min* took 23.4 h (13.3 h), dd*max* took 19.0 h (7.5 h), i.e. DD** in sum took 26% of CPU time of the baseline (46% of wall time).

B. USB drivers of Linux kernel

In the second experiment, 284 modules of Linux operating system kernel USB device drivers, version 5.10.27, were checked for memory safety (no leaking memory, no incorrect dereference, no double free). It was carried out using Klever system [29] on an 8-core Intel Xeon E3-12xx v2 (Ivy Bridge, IBRS) machine with 32 GB of RAM, and a 64-bit Debian 4.9.246-2 OS.

Baseline analysis configuration was (`-smg-ldv`). DD** configuration run same analysis with time limit of 350 s for each verification round.

Fig. 2 shows a quantile graph of the spent CPU time; baseline analysis found 62 *unsafes* (13 of them reqired more than 5 minutes of CPU time), and found 90 *safes* (16 of them required more than 5 minutes of CPU time). Verdict was not produced (result is unknown) for other 132 modules:

- for 5 modules, due to encountered recursive functions in module;
- for 100 modules, because of timeout;
- for 6 modules, because more memory was needed;
- for 21 modules, verification was not conducted at all due to a problem outside of verification tool (these are not shown on the figure).

It can be seen that for modules whose verification takes 15–35 s, the time for the proposed algorithms will most likely also be 15–35 s; the time for modules, the usual verification of which requires more than 35 s, averages 40–50 minutes for dd*max* and 40–90 minutes for dd*min*. Difference under first 350 s is explained by the fact that DD** algorithms do not stop after first error found, while baseline analysis does.

The results for the Linux drivers are presented in the tables II and III. dd*max* and dd*min* obtained 74 and 75 *safe* verdicts, respectively, in cases where verification took less than 350 seconds of CPU time. There was not enough time to

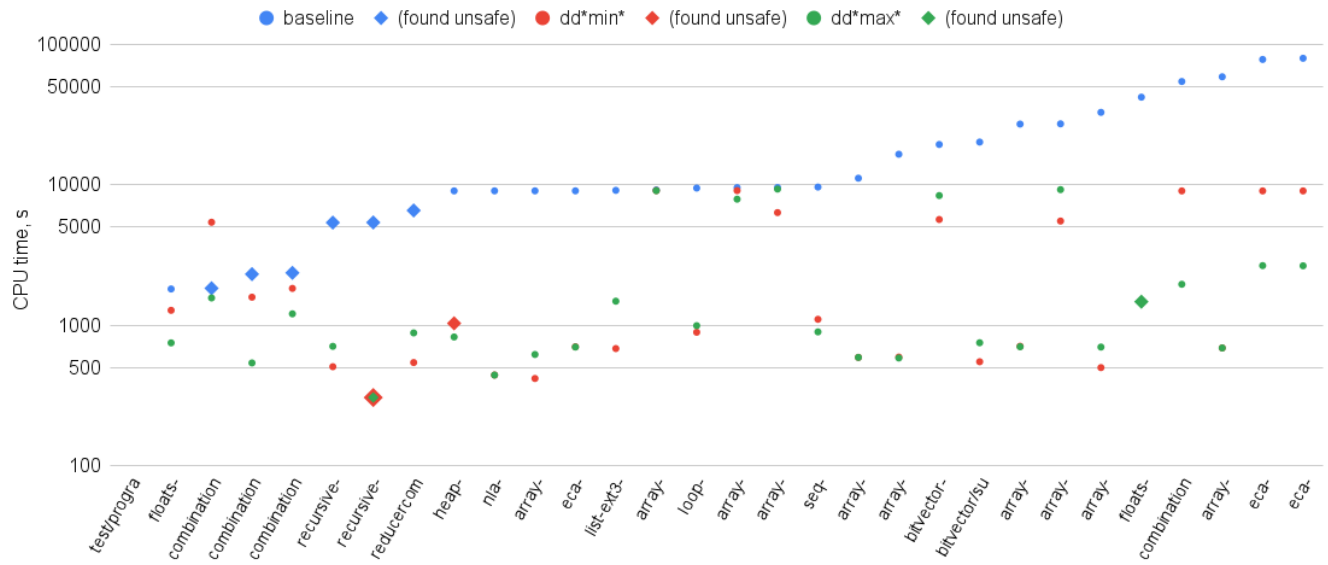

Figure 1. CPU time for analysis of a few benchmark programs (sorted by baseline time)

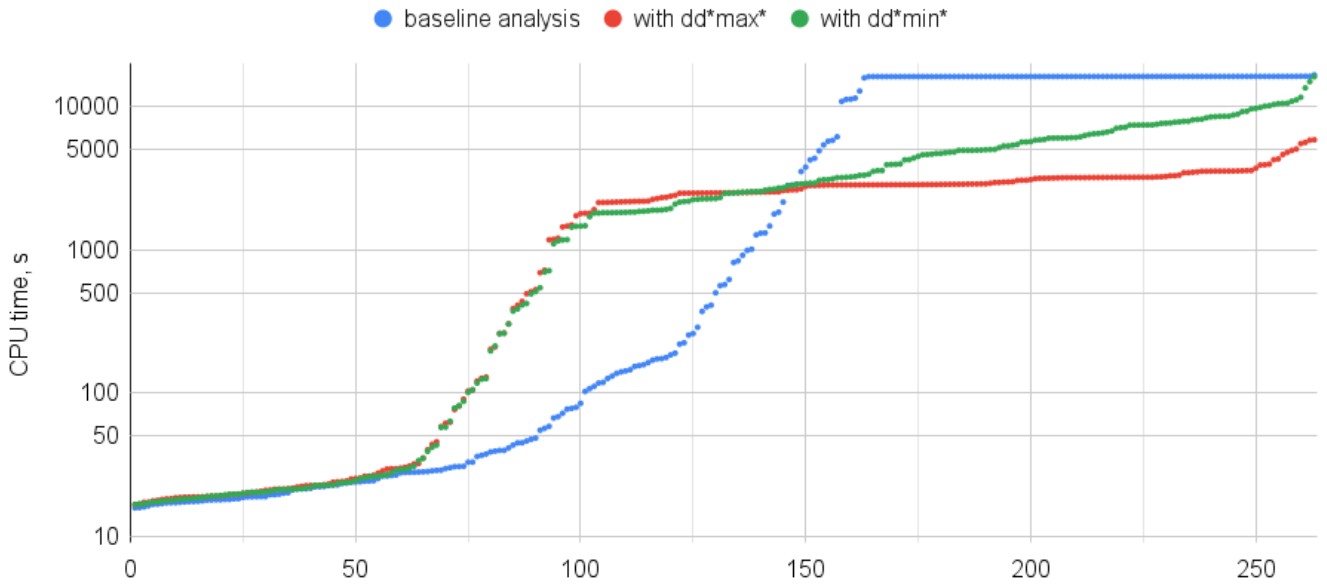

Figure 2. CPU time for analysis of Linux device driver modules (quantile graph)

Table III
Changed verdicts for Linux device drivers

Baseline analysis	dd*max*			dd*min*		
	safe	*unsafe*	*unk.*	*safe*	*unsafe*	*unk.*
safe, 90 in total	74	3	13	75	9	6
unsafe, 62 in total	0	20	42	0	30	32
unknown, 132 in total	0	26	106	0	38	94

verify 100 modules by baseline analysis; there was not enough time for 1 module to analyze using dd*min*. For dd*max* and dd*min*, the analysis of 130 and 50 modules, respectively, ended after dd** enumeration ended without a verdict.

The dd*max* algorithm consumed just 29% of the total CPU time (31% of the total wall time) of the baseline. 26 *unsafes* (42% as percentage of *unsafes* obtained by baseline analysis) were found in programs for which baseline analysis can not obtain a verdict.

The dd*min* algorithm spent 49% of the total CPU time (51% of the total wall time) of the baseline analysis and found 38 *unsafes* (61% as percentage of *unsafes* by baseline

analysis) in modules for which baseline analysis can not obtain a verdict.

In total, DD** algoritms obtained 42 new *unsafes* (23 *unsafes* obtained by both algorithms) for 132 modules with *unknown* baseline verdict.

Change of *safe* to *unsafe* and raise of exceptions can be explained by incorrect counterexample translation: baseline analysis does not stop after target state encoded as specified function call is reached, and C enum types are translated incorrectly.

From the results of the experiments, we can conclude that it may be more effective to use the proposed technique together with a trivial increase of the time limit. For example, simply running the proposed algorithms after the baseline analysis, it is possible to get a linear increase in the number of *unsafes* found (according to the results of the second experiment, 32% of new *unsafes* for additional 29% of total CPU time).

V. Conclusion

In this paper, the problem of software model checking from the point of view of resource constraints is considered. Modern methods and approaches for verification of program models were considered. The problem of finding *unsafes* in programs by simplifying the verified program is stated.

Two algorithms for enumerating simplified versions of programs based on the Delta Debugging algorithms were proposed, implemented in the static verification framework CPAchecker, and evaluated on a small set of programs from SV-COMP benchmark and whole set of 5.10 Linux kernel USB device driver modules.

Experiments have shown that the proposed method, on the one hand, takes less than half the time of baseline analysis and is able to find *unsafes* in programs that are too difficult for baseline analysis, although the total number of verdicts obtained may be less than that of baseline analysis.

As a further work, it is proposed to a) implement the manipulation of program blocks or statements, b) work with error evidence, c) reuse the accuracy obtained in the analysis of the original program, d) study the optimal time for one round of verification and the optimal order of causes.

Acknowledgment

The author thanks Anton Vasilyev and Vadim Mutilin for advice on the article.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*, 2009.

[3] D. Beyer, S. Gulwani, and D. A. Schmidt, *Combining Model Checking and Data-Flow Analysis*. Springer, 2018, pp. 493–540.

[4] V. V. Kuliamin, A. K. Petrenko, and A. V. Khoroshilov, "Component-based verification of operating systems," *Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS)*, vol. 30, no. 6, pp. 367–382, 2018, (in Russian).

[5] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 12652, pp. 401 – 422, 2021.

[6] ——, "Progress on software verification: SV-COMP 2022," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2022.

[7] ——, "Competition on software verification and witness validation: SV-COMP 2023," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2023.

[8] N. Piterman and A. Pnueli, "Temporal logic and fair discrete systems," in *Handbook of Model Checking*, 2018.

[9] A. V. Khoroshilov, M. U. Mandrykin, and V. S. Mutilin, "Introduction to CEGAR — counter-example guided abstraction refinement," *Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS)*, vol. 24, 2013, (in Russian).

[10] D. A. Peled, "Partial-order reduction," in *Handbook of Model Checking*, 2018.

[11] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, "Symmetry reductions in model checking," in *International Conference on Computer Aided Verification*, 1998.

[12] S. Chaki and A. Gurfinkel, "BDD-based symbolic model checking," in *Handbook of Model Checking*, 2018.

[13] A. Biere and D. Kröning, *SAT-based model checking*. Springer, 2018, pp. 277–303.

[14] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[15] M. Chalupa and J. Strejček, "Evaluation of program slicing in software verification," in *International Conference on Integrated Formal Methods*, 2019.

[16] P. Andrianov, V. S. Mutilin, M. U. Mandrykin, and A. A. Vasilyev, "CPA-BAM-Slicing: Block-abstraction memoization and slicing with region-based dependency analysis (competition contribution)," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2018.

[17] M. Spiessl, "Configurable software verification based on slicing abstractions," Master's thesis, Ludwig-Maximilians-Universität München (LMU Munich), München, Germany, Jun. 2018.

[18] F. Nejati, A. A. A. Ghani, N. K. Yap, and A. B. Jafaar, "Handling state space explosion in component-based software verification: A review," *IEEE Access*, vol. 9, pp. 77 526–77 544, 2021.

[19] S. Apel, D. Beyer, V. O. Mordan, V. S. Mutilin, and A. Stahlbauer, "On-the-fly decomposition of specifications in software model checking," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido, "Extreme model checking," in *Theory and Practice*, 2003.

[21] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler, "Precision reuse for efficient regression verification," in *ESEC/FSE 2013*, 2013.

[22] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: a technique to pass information between verifiers," in *SIGSOFT FSE*, 2012.

[23] D. Beyer and S. Kanav, "CoVeriTeam: On-demand composition of cooperative verification systems," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2022.

[24] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, pp. 183–200, 2002.

[25] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," *Proceedings of the 28th international conference on Software engineering*, 2006.

[26] D. Vince, R. Hodován, D. Bársony, and Á. Kiss, "The effect of hoisting on variants of Hierarchical Delta Debugging," *Journal of Software: Evolution and Process*, vol. 34, 2022.

[27] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs," *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[28] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and T. Lemberger, "Verification witnesses," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, pp. 1 – 69, 2022.

[29] E. Novikov and I. S. Zakharov, "Towards automated static verification of GNU C programs," in *Ershov Informatics Conference*, 2017.