DOI: 10.15514/ISPRAS-2023-35(3)-11

Finding More Bugs with Software Model Checking using Delta Debugging

[1,2] *O.M. Petrov*, ORCID: 0009-0004-6245-9615 <*o.petrov@ispras.ru*>

[1] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[2] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

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 might be used to find new bugs in real software.

Keywords: formal software verification; software model checking; delta debugging; CPAchecker.

For citation: Petrov O.M. Finding More Bugs with Software Model Checking using Delta Debugging. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 3, 2023. pp. 151-162. DOI: 10.15514/ISPRAS-2023-35(3)-11

Acknowledgements. The author thanks his colleagues Anton Vasilyev and Vadim Mutilin for their useful advices on the article topic.

Поиск новых ошибок методом верификации моделей с помощью подхода дельта-отладки

[1,2] *О.М. Петров,* ORCID: 0009-0004-6245-9615 <*o.petrov@ispras.ru*>

[1] *Московский государственный университет имени М.В. Ломоносова,*
Россия, 119991, Москва, Ленинские горы, д. 1.
[2] *Институт системного программирования им. В.П. Иванникова РАН,*
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Зачастую инструмент формальной верификации моделей программ не может получить вердикт за ограниченное время из-за комбинаторного взрыва пространства состояний. Чтобы найти ошибки в верифицируемой программе за выделенное время, может быть проанализирована упрощённая её версия. В этой работе представлены алгоритмы DD**, основанные на подходе Delta Debugging, с помощью которых производится перебор упрощённых версий программы. Эти алгоритмы были реализованы в инструменте статической верификации программ CPAchecker. Наши эксперименты показали, что предложенный метод может быть использован для нахождения ошибок в программных системах, используемых на практике.

Ключевые слова: формальная верификация программ; верификация моделей; delta debugging; CPAchecker.

Для цитирования: Петров О.М. Поиск новых ошибок методом верификации моделей с помощью подхода дельта отладки. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 151–162 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–11.

Благодарности. Автор благодарит своих коллег А.А. Васильева и В.С. Мутилина за советы по теме статьи.

1. 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 or binary code of a program without running the program. Lightweight static analysis techniques such as control flow analysis and data flow analysis are 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[2] [4], [5].

The tool is actively developed and wins medals in the software verification competitions SV-COMP several years in a row [6]–[8].

Although at the SV-COMP 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 1 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.

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 to 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, algorithms based on the Delta Debugging algorithm are proposed. The implementation manipulates (removes) function bodies from the internal representation of the program in CPAchecker, a control flow automation.

The proposed enumeration of simplified program versions takes a significant amount of time, and the technique's limitations lead to the loss of up to 38%[3] of verdicts that the baseline analysis could find. However, this way it is possible to get an *unsafe* verdict for the 32% of the programs, for which respective baseline analysis can not obtain a verdict in the same amount of 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://gitlab.ispras.ru/verification/cpachecker
[2] http://linuxtesting.org/ldv
[3] See evaluation on Linux USB drivers in section 4.2.

2. Related work

The following two subsections describe techniques that can be applied in model checking in order to obtain results: specific to the problem of combinatorial explosion in model checking, general-purpose techniques for reduction of the software to be verified, and reuse of partial results of verification. The third subsection describes Delta Debugging approach that is used to enumerate simplified versions of the program to be verified.

Table 1. 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.

2.1 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 [9]). Model checker explores state space of the given program and checks seen states against the given specification. The program state represents values of all program variables and the current control location (the value of the 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 ability of systematic search for error paths makes model checkers useful 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 [10]. 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 verified software, it is used to make the abstraction more precise. The abstract model is refined this way until a feasible counterexample is found or the 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 [11]), and symmetry reduction (using symmetry in systems with multiple identical components [12]), both of which are used for model checking of concurrent systems; and symbolic model checking, i.e. using binary decision diagrams as compact encoding of state space [13].

Another well-known technique is bounded model checking [14]. In order to avoid state-space explosion, the length of explored traces in the model is bounded, and therefore model checker either provides a counterexample that is shorter than the imposed limit, or proves that there are no such counterexamples. This technique is thoroughly improved and is used in practice for bug-finding.

2.2 Partial verification and verification of parts

Another way for state space reduction is to reduce the input program that needs to be modelled. This can be done using component-based approach or reusing previous verification results.

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 [15]. Contrarily, decomposition of specification can also be useful [16]. Incremental verification [15] and extreme model checking [17] 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 [18]. 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 [19] 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 [20]. At the SV-COMP 2022 competition [7], 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.

Another well-known approach that can be viewed as program simplification technique is program slicing [21]: only statements that affect values of the given variables at the given instructions through control or data flow remain in program. This technique was evaluated with CPAchecker [22], [23] with mixed results, and was implemented [24] as a *configurable program analysis* inside CPAchecker (i.e. it can be used alongside other CPA to construct and refine an abstract model of a given program [3]).

2.3 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 [25]. 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 cannot 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 cannot succeed and fail simultaneously). The case when neither is held is considered *unresolved* (here, an error occurred in the verification tool). Seminal work proposes three DD algorithms based on 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 algorithms 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 more effective than *ddmin* [26]. This algorithm uses *ddmin* to minimize each level of the input tree, starting from the root, and removes 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 [27] and binary reduction of dependency graphs (e.g. applicable for Java classes) [28].

3. 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. Accounting for both of these problems, we need to mutate original program until an *unsafe* occurs; then the resulting counterexample is checked against the restored control flow automaton. If the *unsafe* is 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 Delta Debugging)[4].
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*.

3.1 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 coarser elements of the input program before removing more fine-grained elements like blocks and statements, and this case has been implemented and evaluated.

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

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. 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 the 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.

3.2 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. Similarly, 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).

3.3 Counterexample check

CPAchecker has three implementations for checking counterexamples: using CBMC (Bounded Model Checker for C and C++ programs[5]), 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 [29]. 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 may 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.

4. Evaluation

Two experiments 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.

4.1 A few programs from SV-COMP/ReachSafety

29 programs were chosen arbitrarily for the first experiment from ReachSafety category of the SV-COMP benchmark[6]. These programs are checked for reachability of specified function call (reachable call is considered a bug). 21 of the chosen programs have an error (the call is reachable) and 8 of the programs do not have an error (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 in the 2022 competition due to timeout (15 minutes of CPU time).

The time limit was increased from 15 minutes (900 seconds) to 2.5 hours (9000 seconds) of CPU time for verification of one program. The run was performed using BenchExec[7] 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.

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

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

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

Baseline configuration (`-svcomp22 -benchmark` with extended timelimit) uses sequential combination of different analyses [30]. *dd*min** and *dd*max** configurations used same analyses with time limit of 200 seconds for each verification round.

As seen in *Fig. 1* and *Table 2*, 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*. For one program with error, an *unsafe* was found by all three configurations. For another program with error, only *dd*min** found an *unsafe*. For yet another one program with error, only *dd*max** found an *unsafe*.

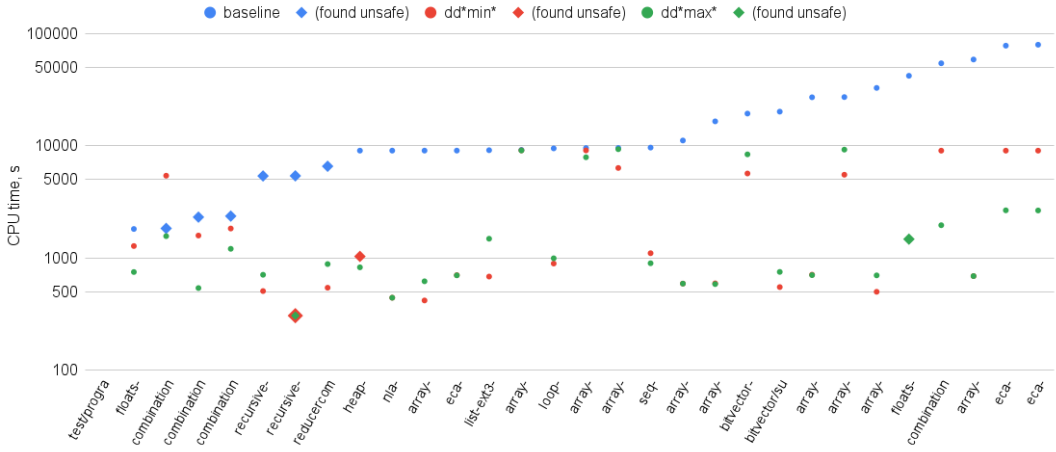

Fig. 1. CPU time for analysis of a few benchmark programs (sorted by baseline time)

Table 2. Results for 29 ReachSafety programs.

	Baseline analysis	***dd*max****	***dd*min****
Total CPU time, h	161	19.0	23.4
Total wall time, h	44.8	7.5	13.3
Safe (8 expected)	0	0	0
Unsafe (21 exp.)	6	2	2
Enumeration completed	—	27	27
Timeout	23	0	0

Small amount of obtained baseline verdicts is not unexpected, as the programs were chosen because CPAchecker could not verify them in time in competition. As these programs consist of small amount of functions, DD** algorithms need more granular elements to manipulate in order to simplify program more precisely and not lose a verdict.

As shown in the table, *dd*min** and *dd*max** in sum took 26% of CPU time of the baseline analysis (46% of wall time).

4.1 Linux USB drivers

In the second experiment, 284 modules of Linux operating system kernel USB device drivers, version 5.10.27, were verified against memory leaks, incorrect dereferences and use after free. It was carried out using Klever system [31] 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 (`-smg-ldv`) uses symbolic memory graphs [32].

*dd*min** and *dd*max** configurations used same analysis with time limit of 350 seconds 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).

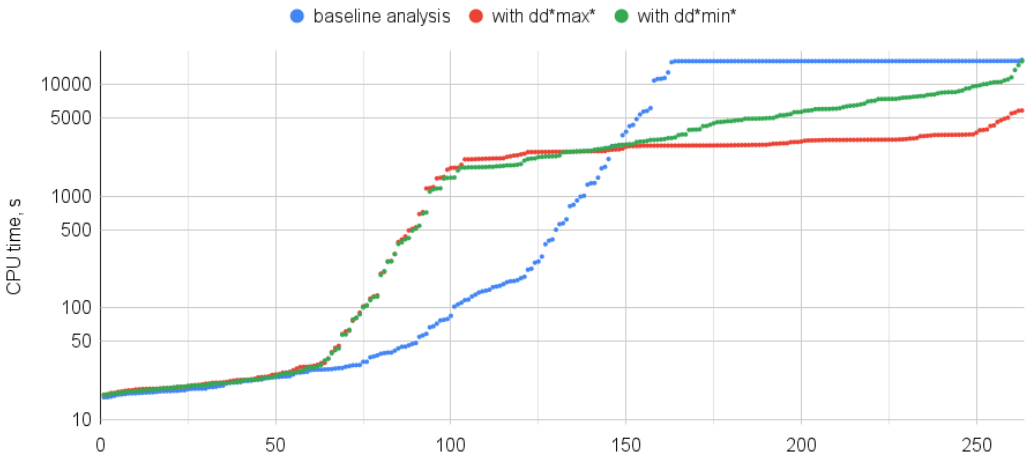

Fig. 2. CPU time for analysis of Linux device driver modules (quantile graph)

It can be seen that for modules whose verification takes 15–35 seconds, the time for the proposed algorithms will most likely also be 15–35 seconds; the time for modules with baseline analysis longer than 35 seconds averages 40–50 minutes for *dd*max** and 40–90 minutes for *dd*min**. Difference under first 350 seconds is explained by the fact that DD** algorithms do not stop verification after first error found, while baseline analysis does. This change in analysis was introduced in order to find all errors that can be present in the original program.

The results for the Linux drivers are presented in *Table 3* and *Table 4*. *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 verify 100 modules by baseline analysis; there was not enough time for one module to analyze using *dd*min**. For *dd*max** and *dd*min**, the analysis of 130 and 50 modules, respectively, ended because enumeration of simplified versions of the module 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 new *unsafes* for 42 modules out of 132 modules with *unknown* baseline verdict. Both algorithms obtained an *unsafe* for 23 of these modules.

Change of *safe* to *unsafe* can be explained by incorrect counterexample check: the used analysis does not stop after target state is reached. Additionally, incorrect translation of C enum types induces raise of exceptions.

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).

Table 3. Results for 29 Linux USB drivers.

	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
Recursion in module	5	5	5
Other exceptions	0	7	46
Other problems	21	21	21

Table 4. Changed verdicts for Linux USB drivers.

Baseline analysis	***dd*max****			***dd*min****		
	safe	*unsafe*	*unknown*	*safe*	*unsafe*	*unknown*
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

5. Conclusion

In this paper, the problem of software model checking is considered from the point of view of resource constraints.

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, *dd*min** and *dd*max**, were proposed for enumerating simplified versions of programs based on Delta Debugging approach. These algorithms were 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 technique takes less than half the total 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.

There are several directions for a future work: a) program blocks and statements manipulation, b) improvement of counterexample translation, c) reuse of partial results obtained in the analysis of the original program or its simplified versions, d) the optimal time for one round of verification, and e) the optimal order of functions and causes in DD** enumeration.

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 Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. Springer, 2011, pp. 184–190.

[3]. D. Beyer, S. Gulwani, and D. A. Schmidt, Combining Model Checking and Data-Flow Analysis. in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, pp. 493–540.

[4]. A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov, “Establishing linux driver verification process,” in Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers 7. Springer, 2010, pp. 165–176.

[5]. I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov, “Configurable toolset for static verification of operating systems kernel modules,” Programming and Computer Software, vol. 41, pp. 49–64, 2015.

[6]. D. Beyer, “Software verification: 10th comparative evaluation (SVCOMP 2021),” Tools and Algorithms for the Construction and Analysis of Systems, vol. 12652, pp. 401 – 422, 2021.

[7]. “Progress on software verification: SV-COMP 2022,” in International Conference on Tools and Algorithms for Construction and Analysis of Systems, 2022.

[8]. “Competition on software verification and witness validation: SVCOMP 2023,” in International Conference on Tools and Algorithms for Construction and Analysis of Systems, 2023.

[9]. N. Piterman and A. Pnueli, Temporal Logic and Fair Discrete Systems, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, p. 27–73.

[10]. A. V. Khoroshilov, M. U. Mandrykin, and V. S. Mutilin, “Introduction to CEGAR — counter-example guided abstraction refinement”, Trudy ISP RAN/Proc. ISP RAS, vol. 24, 2013, (in Russian).

[11]. D. A. Peled, Partial-Order Reduction, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018.

[12]. 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.

[13]. S. Chaki and A. Gurfinkel, BDD-Based Symbolic Model Checking, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, p. 219–245.

[14]. A. Biere and D. Kröning, SAT-based model checking, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, ch. 10, pp. 277–303.

[15]. 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.

[16]. 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.

[17]. T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido, “Extreme model checking,” in Theory and Practice, 2003.

[18]. D. Beyer, S. Lo¨we, E. Novikov, A. Stahlbauer, and P. Wendler, “Precision reuse for efficient regression verification,” in ESEC/FSE 2013, 2013.

[19]. 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.

[20]. 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.

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

[22]. M. Chalupa and J. Strejček, “Evaluation of program slicing in software verification,” in International Conference on Integrated Formal Methods, 2019.

[23]. P. Andrianov, V. Mutilin, M. Mandrykin, and A. Vasilyev, “CPA-BAM-Slicing: Block-abstraction memoization and slicing with region-based dependency analysis,” in Tools and Algorithms for the Construction and Analysis of Systems, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 427–431.

[24]. M. Spiessl, “Configurable software verification based on slicing abstractions,” Master’s thesis, Ludwig-Maximilians-Universita¨t Mu¨nchen (LMU Munich), Mu¨nchen, Germany, Jun. 2018.
[25]. A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” IEEE Trans. Software Eng., vol. 28, pp. 183–200, 2002.
[26]. G. Misherghi and Z. Su, “HDD: hierarchical delta debugging,” Proceedings of the 28th international conference on Software engineering, 2006.
[27]. 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.
[28]. 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.
[29]. 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.
[30]. M. Dangl, S. Löwe, and P. Wendler, “CPAchecker with support for recursive programs and floating-point arithmetic,” in Tools and Algorithms for the Construction and Analysis of Systems, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 423– 425.
[31]. E. Novikov and I. Zakharov, “Towards automated static verification of GNU C programs,” in Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11. Springer, 2018, pp. 402–416.
[32]. A. A. Vasilyev and V. S. Mutilin, “Predicate extension of symbolic memory graphs for the analysis of memory safety correctness,” Programming and Computer Software, vol. 46, pp. 747 – 754, 2020.

Информация об авторах / Information about authors

Олег Максимович ПЕТРОВ — старший лаборант, магистр факультета вычислительной математики и кибернетики (2023). Его научные интересы включают верификацию моделей программ, delta debugging.

Oleg Maximovich PETROV is a senior laboratory assistant and a master of the Faculty of Computational Mathematics and Cybernetics (2023). His research interests include software model checking, delta debugging.