ISSN 0361-7688, Programming and Computer Software, 2016, Vol. 42, No. 4, pp. 225–238. © Pleiades Publishing, Ltd., 2016.
Original Russian Text © V.O. Mordan, V.S. Mutilin, 2016, published in Programmirovanie, 2016, Vol. 42, No. 4.

Checking Several Requirements at Once by CEGAR

V. O. Mordan and V. S. Mutilin

Institute for System Programming, Russian Academy of Sciences,
ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: mordan@ispras.ru, mutilin@ispras.ru

Received November 15, 2015

Abstract—Currently, static verifiers based on counterexample-guided abstraction refinement (CEGAR) can prove correctness of a program against a specified requirement, find its violation in a program, and stop analysis or exhaust the given resources without producing any useful result. Theoretically, we could use this approach for checking several requirements at once; however, finding of the first violation of some requirement or exhausting resources for some requirement will prevent from checking the program against other requirements. Moreover, if the program contains more than one violation of the requirement, CEGAR will find only the very first violation and may miss potential errors in the program. At the same time, static verifiers perform similar actions during checking of the same program against different requirements, which results in waste of a lot of resources. This paper presents a new CEGAR-based method for software static verification, which is aimed at checking programs against several requirements at once and getting the same result as basic CEGAR checking requirements one by one. To achieve this goal, the suggested method distributes resources over requirements and continues analysis after finding a violation of a requirement. We used Linux kernel modules to conduct experiments, in which implementation of the suggested method reduced total verification time by five times. The total number of divergent results in comparison with the base CEGAR was about 2%. Having continued the analysis after finding the first violation, this method guarantees that all violations of given requirements are found in 40% of cases, with the number of violations found being 1.5 times greater than in that in the base CEGAR approach.

DOI: 10.1134/S0361768816040058

1. INTRODUCTION

Static verification is a formal means for checking program source code without its execution by exploring all possible program paths. The main benefit of static verification is that it aims at proving correctness of the software instead of simply searching frequent bugs. The main disadvantage, which hampers its use in practice, is a large amount of required resources such as CPU time and memory, especially for large-scale software systems. Currently, static verification tools (static verifiers) based on Counterexample Guided Abstraction Refinement (CEGAR) can be applied to large-scale software systems, which is demonstrated in annual Competitions on Software Verification (SV-COMP) [1, 2].

At the same time, every program in a big software system may contain any number of different bugs, which are violations of requirements to that program. In order to find them or prove that they are lacking, static verifiers check the program against all specified requirements. The base CEGAR approach stops after finding the first bug assuming that the program is not correct and there is no sense to further check it; i.e., the base CEGAR is capable of finding only the first violation of a requirement. In addition, given that it is impossible to prove program correctness against one requirement, it will also be impossible to prove its correctness against several requirements. Therefore, in practice, CEGAR is used only for checking a program against one requirement at a time.

This results in unjustified waste of resources. On the one hand, the amount of resources required for static verification increases proportionally to the number of requirements to be verified. On the other hand, to find all bugs in a program that violate a given requirement, the static verifier should be launched as many times as this requirement is violated and eliminate all violations found.

As an example, we consider the configurable static verification system Linux Driver Verification Tools (LDV Tools). The LDV Tools system is aimed at checking correctness of Linux kernel modules with the help of different static verifiers [3]. The requirements to be verified in the LDV Tools system are rules describing correct use of program interfaces of the kernel. The system has already helped to find more than 200 bugs in Linux kernel modules, which have further been eliminated [4]. The process of verification of all Linux kernel modules with the help of LDV Tools against a single rule specification by static verifiers

BLAST [5, 6] or CPAchecker [7] takes from two to three days, whereas the verification of the same modules against 17 basic rule specifications takes about 40 days. Note that the number of such requirements on Linux kernel modules may reach several hundred specifications.

This paper presents a new static verification method, which can significantly reduce the verification time even if the number of requirements to be checked and the number of violations of these requirements permanently grow. The suggested method extends CEGAR in order to check more than one requirement at once. The main demand to this method is to prove correctness of programs against specified requirements and to find their violations like CEGAR does, but faster. In addition, the method should provide an opportunity for finding all violations of each requirement being verified. The proposed method was implemented as an extension of LDV Tools [3] and the CPAchecker [7] static verifier and was evaluated on verification of Linux kernel modules.

The contributions of this paper are as follows:

1. A new static verification method, which extends the CEGAR approach to checking a program against several requirements at once, and its modification aimed at finding all violations of all requirements being verified are proposed.

2. The proposed method is implemented as an extension of the LDV Tools system and the CPAchecker static verifier.

3. It has been experimentally shown that the suggested method reduces the verification time by five times, with the total number of negative results (in which the suggested method fails to prove correctness of the program against specified requirement or to find its violation) being about 2%.

4. It has been experimentally shown that the modification of the proposed method aimed at finding all violations of all requirements being verified finds all violations in 40% of all cases (i.e., it is proved that there are no other violations of the considered requirements), with the number of violations found being 1.5 times greater than that found with the help of the basic CEGAR approach

Section 2 describes the basic CEGAR approach and introduces definitions used in the paper. Related works are discussed in Section 3. Section 4 presents the new method and its modifications. An implementation of the proposed method is described in Section 5. Experimental results are presented in Section 6.

2. BACKGROUND

2.1. Definitions

In the paper, a formal representation of a verified requirement to a program is called *assertion*. The assertion describes formally the requirement being verified. The assertion can be represented as auxiliary checks, which are added to the program code (e.g., *assert(condition))*.

Error location is a location in a program that corresponds to an assertion violation (for example, predefined function *error*() or predefined label *ERROR* corresponding to an auxiliary check added). For instance, construct *assert*(*condition*) is opened as follows:

@if (!condition)

ERROR: error(); // error location@

Verification task is a program source code built on a program under analysis containing specific checks for the assertion. In CEGAR, verification tasks are usually reachability tasks; their main goal is to prove that an error location cannot be reached from a specified entry point in the program (for example, from function *main*).

Error trace is a sequence of operations in the program source files that leads from a specified entry point to an error location (i.e., an assertion violation).

Verdict is a result of solving a verification task by a static verifier. Usually, there are three possible verdicts:

• *Safe*: the program is correct against the specified assertion;

• *Unsafe*: the program violates the specified assertion, and the corresponding violation is represented as an error trace;

• *Unknown*: the static verifier cannot solve the given verification task.

Concrete state consists of assignments of specific values to relevant variables at particular program locations [6].

Abstract state is a set of concrete states of the program [6].

Abstract Reachability Graph (ARG) is an abstraction of the program in which nodes correspond to the abstract states and edges correspond to the program statements [6].

Verification fact is a result (possibly intermediate) of the verification process, which determines the current level of abstraction of the program [8].

Precision is a verification fact that instructs the analysis on what information should be tracked and what information should be omitted in the construction of the program abstraction. For example, in the predicate analysis [9], precision determines predicates to be tracked; in the explicit value analysis [10], precision determines variables to be tracked.

2.2. Counterexample Guided Abstraction Refinement

The basic idea of CEGAR is to construct an abstract program model, to iteratively refine it, and to prove unreachability of the error location in the

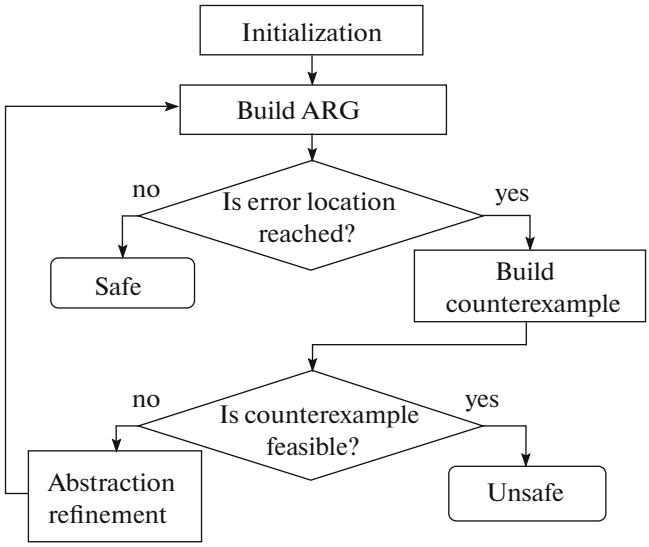

Fig. 1. The CEGAR loop.

abstract model. The CEGAR algorithm is presented in Fig. 1 [11, 12]. First, CEGAR builds ARG based on an initial precision (for example, an empty precision). If a specified error location is not reached, then the algorithm terminates with the verdict Safe. Otherwise, CEGAR checks the counterexample found for feasibility. If it is feasible, then an error trace is built for the counterexample and the algorithm terminates with the verdict Unsafe. Otherwise, the precision is refined based on the infeasible counterexample and the CEGAR loop is continued. Since static verification is an undecidable problem in the general case, static verifiers implementing the CEGAR approach operate with limited resources (such as CPU time and memory) in practice and terminate their analysis with the verdict Unknown if the resources are exhausted.

2.3. LDV Tools-based Example

Suppose that there are three following requirements:

Requirement 1: *all resources allocated by usb_alloc_urb*() *should be freed by usb_free_urb*()

Requirement 2: *the same mutexes should not be acquired or released twice in the same process.*

Requirement 3: *the offset should not be greater than the array size.*

In LDV Tools, verification task is prepared by instrumenting the program source code [13]. Each requirement is formally represented as an assertion. Violation of assertion i may correspond to error location $ERROR\_i$, $i = 1, 2, 3$. For instance, Requirement 3 may be represented in the verification task by means of the following assertion:

```
if (offset > size)
ERROR_3: goto ERROR_3;
```

3. RELATED WORK

Suppose that there are N assertions and it is required to check a program against these assertions using static verification.

3.1. Sequential CEGAR

The most natural way to do this is to sequentially prepare and solve N verification tasks by the basic CEGAR algorithm. In this case, the average verification time increases by N times compared to that by basic CEGAR, and N verdicts will be obtained. However, this method does not take into account the fact one and the same program is verified; therefore, many operations will be repeated. Thus, this is a very simple and safe, but very inefficient, method to solve the posed problem.

3.2. Batch CEGAR

Potentially, it is possible to prepare one verification task by adding to it N assertions and to solve it by the basic CEGAR algorithm. However, in this case, we will obtain only one verdict for all assertions. Here, two difficulties may arise. First, if verdict *Unsafe* is obtained for an assertion, then the CEGAR algorithm stops and no verdicts are obtained for other assertions. Second, if the verification task cannot be solved for at least one assertion (verdict *Unknown*), then it will not be solved for all assertions. In this case, the verification facts obtained for one assertion will prevent from proving correctness of another assertion (for example, if this requires coarser abstraction). The basic cause of this difficulty is that the verification task for N assertions is more complicated than that for one assertion and requires a more detailed abstraction. Thus, this approach leads to significant loss of results compared to sequential CEGAR. Therefore, even theoretically, this method is not capable of providing us with a result that is comparable with that provide by sequential CEGAR.

3.3. Regression Verification

Regression verification is aimed at verifying program revisions to exclude addition of new bugs. In this case, the program being analyzed (and, hence, the verification task) is practically the same. One of the approaches to do it efficiently is to reuse verification results [8]. Verification facts obtained upon verification of the original version of the program are stored and, then, used on the *initialization* step of the CEGAR algorithm (Fig. 1) in the verification of a revised version of the program to start analysis with the known level of abstraction, which can significantly reduce the analysis time. For the verification facts, one can use precision [14], a set of abstract states (ARG), annotation functions, invariants, etc. Experiments confirm [9] that reuse of verification facts

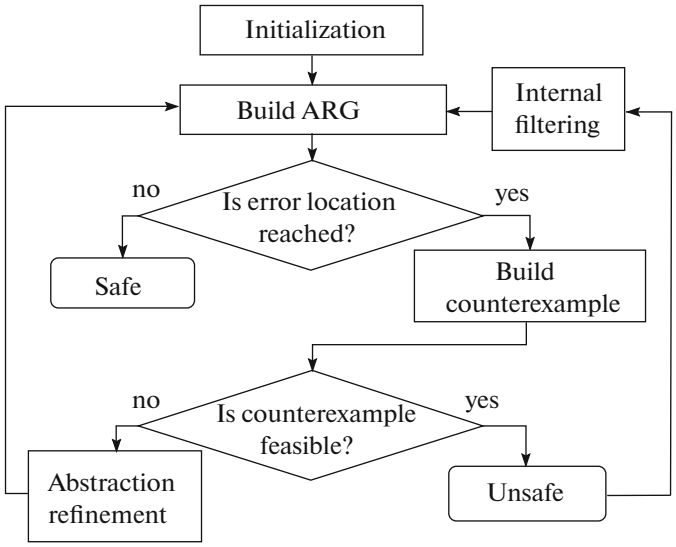

Fig. 2. Algorithm MEA.

reduces time for regression verification by several times. At the same time, in some cases (for example, if program revisions are significant), reuse of verification results may increase time of analysis. For instance, precision reuse for some verification tasks increased time almost twice [14]. Thus, reuse of verification facts can both reduce and increase verification time. Moreover, these methods were earlier applied only for checking different versions of one program rather than for checking different assertions in a program.

3.4. Conditional Model Checking

The conditional model checking [15] is aimed at combining various techniques for resolving verdicts *Unknown*, when the verification task cannot be solved with the use of a single technique. In order to solve this problem, a static verifier saves its result even in the case of unsuccessful termination (e.g., upon exhausting the resources), which describes what parts of the program (e.g., in terms of abstract states) were successfully verified and what parts were not. Then, another static verifier (or the same verifier with a different configuration) takes this result at the *initialization* step of the CEGAR algorithm (Fig. 1) and verifies only those parts of the program that were not verified. Experiments [15] justify that conditional model checking helps to solve problems that cannot be solved by a single launch of a static verifier.

3.5 Analysis of Many Errors

As is known, the basic CEGAR algorithm stops after it finds the first error (Fig. 1). This is because it is assumed that the program that violated the assertion being verified cannot be correct and there is no need to further try to prove its correctness. In order to find more than one bug by using the CEGAR approach in a program violating a given assertion, it is required to iteratively fix the bug found and continue verification until correctness of the program is proved (verdict *Safe* is obtained). However, in practice, such an approach is very inefficient, since it requires solving almost the same verification task many times; moreover, the process of bugs fixing cannot be automated. For example, it happened that fixing of a few bugs of one type in the Linux kernel modules took several months [16].

To solve this problem, multiple error analysis (MEA) [16] was suggested. MEA aims to continue analysis after the basic CEGAR algorithm finds an error, to obtain several error traces in the end, and to filter the error traces found. Clearly, MEA requires more resources than the basic CEGAR algorithm does. The basic difficulty associated with this method is that each error can generate several (in some cases, infinitely many) error traces (paths from the entry point to the error location). Hence, the traces need to be filtered. In the general case, this problem is undecidable, since the concept of "error" cannot be formalized (for example, the error can consist in lacking an action in the error trace or false firing of the static verification tool); therefore, the problem cannot be solved automatically. MEA suggests to automatically reduce the number of "identical" traces as much as possible and to filter them manually if needed.

Light-weight **internal filtering** is added to the algorithm (Fig. 2). It is based on the internal representation of error traces to rapidly exclude evident duplicates. If an error trace found based on some criterion is equivalent to a trace found earlier, it is dropped. Examples of internal filters are comparison of complete equivalence, comparison by program blocks [9], etc. This type of filtering seems to be very efficient and is applicable to any analyzed programs.

Heavy-weight **external filtering** is performed after termination of the algorithm to make filtering maximally complete with regard to assertion representation. The basic idea of this filtering is to extract the so-called kernel for each error trace based on a given transformation function and to compare kernels instead if traces themselves. This type of filtering may require more time and is applicable to only some types of assertions.

In some cases, after automated filtering, there remain several error traces, which require **manual filtering** [16]. The basic goal of manual filtering in MEA is to separate error traces associated with different errors in a minimal number of actions. First, the user analyzes an error trace and finds the error itself. Then, the user specifies a transformation function in order to extract the error trace kernel that characterizes the error found and the **comparison function** to determine how this kernel will be compared with kernels of other error traces (e.g., complete coincidence, inclusion,

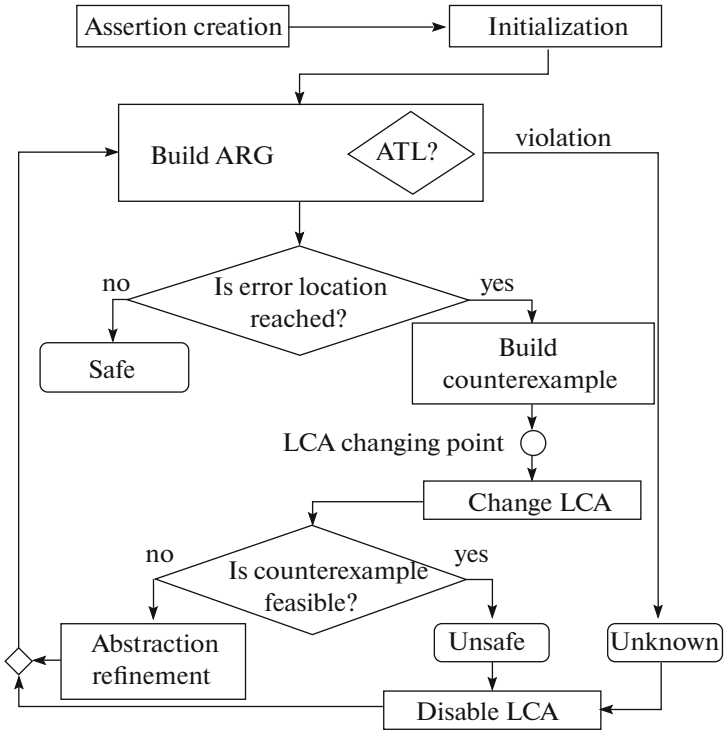

Fig. 3. Algorithm MAV.

etc.). After this, MEA applies the transformation function to all other error traces and compares the kernel of the considered error trace with the kernels of other traces by using the comparison function. The error traces for which the comparison function returns truth are considered to be equivalent to the analyzed one. Thus, the user looks through only those traces that correspond to different errors.

MEA adds the new—**Unsafe-incomplete**—type of verdict. It means that some error traces have been found but the analysis terminated abnormally. For instance, resources have been exhausted (i.e., some errors have possibly been missed).

We carried out an experiment with MEA similar to that in [16]. We took 440 verification tasks (each contains at least one error) with verdict *Unsafe* obtained upon checking kernel modules of Linux (version 3.12-rc1) by LDV Tools against 17 basic assertions. Time of MEA operation increased by about two times compared to basic CEGAR. In more than half cases, verdict *Unsafe* changed to *Unsafe-incomplete*; after manual filtering, 665 errors were found (i.e., 1.5 times greater).

3.6. Conclusions

We have discussed two approaches to solving the posed problem. Sequential CEGAR provides good results but is very inefficient, since information is not reused when different assertions are checked. Batch CEGAR is not appropriate because of verdicts *Unsafe* and *Unknown*. The reuse of verification facts based on the idea of regression verification can both reduce and increase analysis time. Conditional model checking can help to resolve verdict *Unknown*. MEA resolves the problem with verdicts *Unsafe* but creates another problem: there arises necessity in filtering (including manual filtering) of error traces.

4. MULTI-ASPECT VERIFICATION

Multi-aspect verification (MAV) is aimed at configurable checking of several assertions for a given verification task at once. Multi-aspect verification is a versatile program checking, which is supposed to find all violations of all assertions being verified. In this paper, MAV is suggested as an extension of the CEGAR algorithm, but potentially the same ideas can be applied to other static verification approaches.

It is supposed that the verification task has already been built for the selected assertions like in the case of batch CEGAR (for more detail, see Section 5). MAV receives a given task at its input and outputs verdicts for all assertions (like sequential CEGAR). The MAV method should meet the following requirements:

- to get the same verdicts as sequential CEGAR;
- to consume less computational resources (basically, CPU time) than sequential CEGAR does;
- to have a possibility for finding several violations of assertions (like in MEA).

4.1. Basic Multi-Aspect Verification Algorithm

The MAV algorithm was designed to meet the above-mentioned requirements (Fig. 3). It is an extension of the basic CEGAR.

In terms of MAV, each **assertion** contains the following attributes: a unique identifier, the corresponding error location identifier, an assertion verdict, consumed resources (CPU time), and the corresponding verification facts. At the *assertion creation* step, links between assertions and error locations are created, and attributes for each assertion get initial values.

Assertion verdict is an internal verdict of an assertion in the course of the MAV algorithm operation. The assertion verdict may take all common verdict values (which were defined in Section 2) and the auxiliary value *Checking*, which means that this assertion is currently being checked.

In order to know, which assertion is being checked (to change assertion verdicts, to keep records of consumed resources, etc.), the notion of the latest checked assertion (LCA) is introduced. LCA is the latest assertion checked by the algorithm. However, it is quite clear that, in the course of the ARG construction, several assertions can simultaneously be checked (the idea MAV is based on). Therefore, the following approximation is suggested: *only one assertion is being checked at each time moment, and the latest assertion checked is LCA*.

At the start of the algorithm operation, LCA is unset. Then, all time of analysis can be divided by the so-called **LCA changing points**, which are moments of time when LCA changes. Current checked assertion in the framework of the given approximation is considered equal to LCA until the next LCA changing point. Since counterexample always contains the error location reached, it is always possible to determine which assertion it violates. We assume that the time spent by the algorithm for the ARG construction to get the given counterexample should be considered as the time spent on the verification of the assertion violated in the counterexample (if the counterexample is true, then a violation of the assertion was found; otherwise, the abstraction is refined based, among other things, on the "violated" assertion; i.e., we can guarantee that this assertion (possibly, not only this one) was certainly checked). Thus, the LCA changing points can be added immediately after the counterexample construction step in the algorithm.

After that, the entire analysis time is divided into the so-called **LCA intervals**, which are time intervals between two neighboring LCA changing points. Further, we can use the LCA intervals to calculate time spent for each assertion. In order to do this, we add time of the interval to the LCA time at the **LCA change** step. In addition, the verification facts obtained on this step are added to the verification facts of this assertion.

After that, we may constrain time spent for each assertion with the help of the assertion time limit (ATL). If ATL is violated, LCA will be disabled on the *LCA disable* step. Since, in the general case, the LCA interval can exceed ATL by itself, ATL is checked asynchronously.

Therefore, assertion verdicts are changed in the course of analysis in the following way. At the *assertion creation* step, every assertion gets verdict *Checking*. If an error trace for some assertion is found, its assertion verdict changes to *Unsafe*. If ATL is exhausted, the corresponding assertion verdict changes to *Unknown*. If the algorithm stops with verdict *Safe*, then all assertions that have verdict *Checking* get verdict *Safe*.

On the *LCA disable* step, the verification of LCA terminates and the LCA verification facts are removed. After this step, the algorithm continues to operate; however, the disabled assertions will not be further checked. Moreover, the removal of the verification facts corresponding to the disabled assertion will prevent from the negative effect of these facts on the verification of the other assertions.

Thus, the proposed algorithm of multi-aspect verification is capable of checking N assertions and producing N verdicts for them. Owing to the reuse of the verification facts for different assertions, the total verification time should reduce like in the regression verification (similar actions are not repeated like in sequential CEGAR). Owing to the removal of the verification facts on the *LCA disable* step, verification facts of different assertions will not interfere with one another, which is the basic difficulty in batch CEGAR. Theoretically, the verdicts obtained should be identical to those obtained by sequential CEGAR.

4.2. Extensions of the Basic Algorithm

The above-described MAV algorithm can be configured and, if needed, extended depending on the requirements imposed. Moreover, all approximations discussed can be generalized.

Verification facts. In the regression verification, various verification facts model the current abstraction level, which determines efficiency of their reuse. In addition, it should be taken into account how easy verification facts can be passed between different runs of the static verifier. It is also necessary in MAV to determine how the verification facts can be removed on the *LCA disable* step. For instance, the set of abstract states describes an abstraction in the most complete way, and this set can easily be reused (MAV works with one and the same ARG for different assertions). However, the operations of saving and, especially, removing sets of abstract states are very inefficient, since ARG usually consists of millions of abstract states.

Verification facts removal strategy. In the ideal case, negative effect of a disabled assertion disappears com-

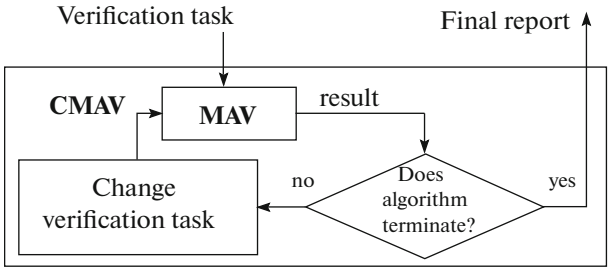

Fig. 4. Algorithm CMAV.

pletely on the *LCA disable* step. However, it should be taken into account that, in the general case, reuse of one or another verification fact can both speed-up (positive effect) and increase (negative effect) time of analysis of other assertions. For example, when getting verdict *Unknown* for some assertion, there may (seldom) arise a situation where the current abstraction level makes it possible to easily prove correctness of other assertions, whereas the removal of verification facts (which takes some time) simply increases the total analysis time. Thus, in practice, this operation should be performed based on analysis of the balance between the positive and negative effects of the verification facts reuse.

Internal time limits. Generally, the earlier proposed ATL can be extended. The basic idea of any time limit is to constrain execution time of some operation and to perform predefined actions if it is violated. For instance, ATL constrains total time of assertion verification and, if it is violated, disables this assertion with verdict *Unknown*. Potentially, we can constrain time of any operation of the algorithm for any LCA interval, etc. For example, if it is known that, if the first LCA interval is longer than a given value (i.e., the program itself is too complicated for verification), then there is no sense to continue analysis, since all assertions eventually will exceed ATL. In this case, one can constrain the first time interval by a prescribed value. Moreover, we may constrain execution time for any operation and disable the assertion that violated the time limit with verdict *Unknown*, assuming that this assertion not only will get verdict *Unknown* in any case but also will impede checking other assertions.

Approximation with LCA. The proposed approximation with a single assertion being verified can be extended to the general case. Then, we will deal with an LCA set, which is to be updated at the LCA changing points. This approximation is much closer to the reality; however, the problem of tracing time and verification facts for each assertion cannot be solved precisely. Depending on the type of analysis, various heuristics can be proposed to solve this problem. For example, in the analysis of explicit values [10], we can use the current set of variables being traced to determine the set of assertions corresponding to them and, any time when this set changes, to perform the LCA change step. In addition, it should be taken into account that change of LCA involves overheads.

Besides extensions considered above, it is theoretically possible to modify the basic MAV algorithm for use beyond the scope of CEGAR.

4.3. Conditional Multi-Aspect Verification

Theoretically, MAV always outputs verdicts Safe, Unsafe, or Unknown for each assertion being verified. However, in practice, this is not always true if the tool implementing the algorithm has not been terminated successfully, for example, when only one assertion resulted in an internal error in the static verifier (or in exhausting available memory). In the latter case, we would like to get verdicts for other assertions, which had verdict Checking (i.e., no verdict in the framework of basic CEGAR) at the moment of the unsuccessful termination of the tool. In order to achieve this, MAV was extended based on the ideas of conditional model checking [15]. The basic idea is to launch the MAV method a few times, passing intermediate results between the launches until all assertions get verdicts. We call this extension conditional MAV (CMAV). The schema of CMAV is presented in Fig. 4.

In order to save the intermediate results to be passed between the launches, we suggest **common CMAV format**, which keeps for each assertion the following attributes: assertion identifier, assertion verdict, time consumed, error trace identifier (for assertions with verdict *Unsafe*), the cause of verdict *Unknown* (for assertions with verdict *Unknown*). In addition, it is necessary to store the LCA identifier. This intermediate result in the common format is stored into a specified file. At the LCA change and LCA disable steps, as well as in the beginning and end of analysis, information in this file is updated. Thus, in the case of any abnormal termination of the tool operation, this file will contain all relevant information about the analysis.

We call each launch of the MAV algorithm an **iteration** of CMAV. After each iteration, CMAV analyzes the result passed in the common CMAV format using the following **CMAV termination algorithm:**

• If the result is lacking, it is assumed that a global error occurred (for example, a syntax error in the program code or incorrect tool configuration), all assertions get verdict *Unknown* and analysis terminates.

• If the number of assertions with verdict *Checking* is 0, then analysis terminates.

• If at least one assertion has verdict *Unsafe* or *Unknown*, then the result for such assertions is stored, they are completely removed from the verification task, and the next iteration is launched.

• If all assertions have verdict *Checking* and LCA is not known (i.e., analysis does not reach the first LCA changing point), it is assumed that a global problem

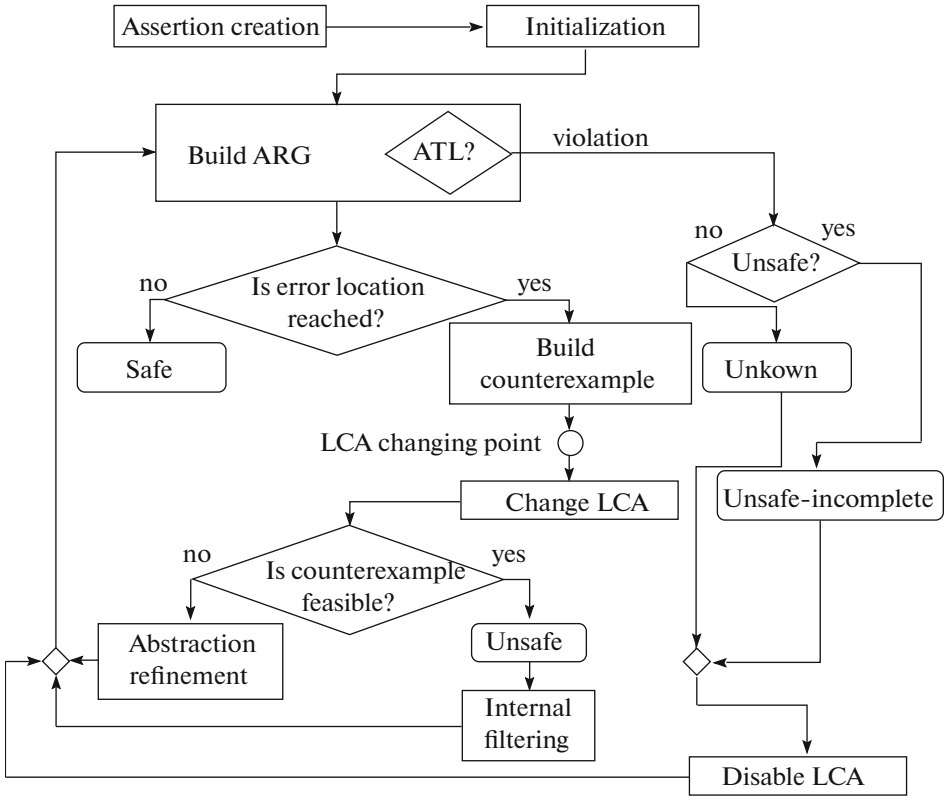

Fig. 5. Algorithm MAV with MEA.

takes place; all assertions get verdict Unknown, and analysis terminates.

• If all assertions have verdict *Checking* and LCA is known, than it is LCA that is considered to be "guilty" of incorrect termination of the algorithm operation. This assertion gets verdict *Unknown* and is removed from the verification task; after this, the next iteration is launched.

After termination of the last iteration, CMAV combines intermediate results from all iterations and forms a final report, which contains pairs assertion–verdict and error traces for all verdicts *Unsafe*.

Thus, the CMAV method is capable of finding verdicts for N assertions in not more than N iterations under the condition of constrained resources and, possibly, incorrect termination of static verifier operation.

4.4. Integration with MEA

Now, we consider a more general problem: find all violations of given assertions or prove their correctness. All tools for solving this problem—the CMAV and MEA methods—are already available. The CMAV method is capable of finding one violation for each assertion being verified or proving their correctness. MEA is aimed at finding all violations of one assertion.

To solve the posed problem by the basic MAV algorithm, it is required to continue analysis after finding a violation of the assertion and not to disable this assertion (Fig. 5). In addition, similar to MEA, internal filtering of error traces is needed in this place. The error traces that passed internal filtering are saved in the attributes of the corresponding assertion (to be used for internal filtering of other traces) and, immediately after this, are written into a file (because the analysis can terminate incorrectly), with the assertion verdict being changed into Unsafe. If ATL is violated, the assertion is disabled, and its verdict changes into *Unsafe-incomplete* if it earlier was *Unsafe* or into *Unknown* otherwise (Fig. 5).

In the CMAV termination algorithm, it is required to modify Step 3 as follows. If there are verdicts *Unsafe* and *Checking*, then all verdicts *Unsafe* are changed into *Unsafe-incomplete* (the algorithm did not terminate correctly; possibly, not all error traces have been found). If at least one assertion had verdict *Unsafe*, *Unsafe-incomplete*, or *Unknown*, external filtering of the error traces found is performed, the filtered traces are stored, the corresponding assertions are completely removed from the verification task, and the next iteration is launched. The final report will contain pairs assertion–verdict and error traces for the assertions with verdicts *Unsafe* and *Unsafe-incomplete*.

Table 1. Comparison of cleaning adjustable precision strategies (for sequential CEGAR, iterations are meant to be sums of all launches of the basic CEGAR algorithm, l is the average number of iterations required for solving one verification task)

Strategy	CPU time (hours)	Error traces	Iterations	l
CMAV: None	140	543	2084	2.084
CMAV: WL/Subtract	118	610	1984	1.984
CMAV: WL/Clear	127	599	1580	1.58
CMAV: ARG/Subtract	122	590	2005	2.005
CMAV: All	120	605	1451	1.451
Sequential CEGAR	389	604	17000	17

Thus, the method of conditional multi-aspect verification with analysis of many errors (CMAV with MEA) guarantees finding of all violations of a given assertion for verdicts *Unsafe* and some (possibly, not all) violations for verdicts *Unsafe-incomplete*.

5. IMPLEMENTATION

We have implemented the suggested method as extensions of the LDV Tools system and the CPAchecker static verifier. The LDV Tools system prepares verification tasks based on given multiple assertions; launches CMAV iterations; performs external filtering for MEA; prepares the final report; and provides means for analyzing results, including those for manual MEA filtering. The CPAchecker static verifier executes the MAV algorithm.

5.1. LDV Tools Extensions

LDV Tools take multiple assertions as input and join them into a combined assertion. The combined assertion contains references to the original assertions. In our implementation, each error location contains a unique identifier of the corresponding assertion. For combining the assertions, the LDV Tools system was equipped with a new component.

The CPAchecker input configuration includes a set of error labels, which contain references to the original assertions. By modifying this set, we modify the verification task selecting what assertions should be checked by the static verifier.

After termination of each iteration, the LDV Tools system gets at its input the result in the common CMAV format and applies the CMAV termination algorithm to it in order to determine whether the next iteration is needed. If the analysis is not complete, the result obtained in the course of the iteration is stored, the verification task changes by changing the set of assertions being verified in the configuration of the CPAchecker static verifier, and the next iteration is launched. Otherwise, the LDV Tools system prepares final report, which contains, together with each assertion, its verdict and, possibly, the error trace.

In the case of MEA, before saving error traces, LDV Tools performs external filtering. To this end, it uses a filter of assertion model functions [16]. The assertion model functions describing the assertion logic in LDV Tools are added into the verification task in the course of instrumentation process. The core of this filter is calls of model functions.

5.2. CPAchecker Extensions

We extended the CPAchecker static verifier in order to support the MAV method and its extensions.

Assertion. CPAchecker takes the configuration from LDV Tools, which contains the set of error labels verified (those containing assertion identifiers). Based on this information, pairs error identifier–error label are created on the *assertion creation* step. On the LCA disable step, CPAchecker stops checking the corresponding error label.

Verification facts. For the verification facts, we use abstract states and adjustable precision. Abstract states are not saved for each assertion on the *LCA change* step and are not removed on the *LCA disable* step, since these operations are very inefficient. However, it may happen that ARG is too complicated to prove correctness of some assertions (for example, 99% of all abstract states correspond to the assertion that violates ATL). Therefore, in such cases, it is simpler and faster to completely rebuild ARG. To this end, it is suggested to launch a new CMAV iteration (see further for more detail). In the course of the entire CMAV iteration, abstract states are reused for all assertions being verified.

Adjustable precision introduces new (compared to those for conventional precision [14]) operations of addition, subtraction, and clearing. On the *LCA change* step, we save precision obtained in the course of abstraction refinement and add it to the LCA precision. Adjustable precision is reused with all assertions until the *LCA disable* step for the given assertion or until termination of the CMAV iteration.

Internal time limits. One possible solution of the ARG complexity problem is to launch a new iteration. To this end, we suggest using idle interval time limit (IITL). An idle time interval is time from the *LCA dis-*

Table 2. Comparison of verdicts in sequential CEGAR and CMAV for verification of all Linux kernel modules of version 4.0-rc1 (6021 verification tasks) against 17 assertions

Algorithm	*Safe*	*Unsafe*	*Unknown*
Sequential CEGAR	98651	623	3083
	−2195 (2.15%)	−22 (0.02%)	
CMAV	96654	624	5079
	+198 (0.20%)	+23 (0.02%)	

Table 3. Results of automated filtering of error traces found by the CMAV with MEA method

Type of filtering	Error traces
Without filtering	531123
Internal filtering	31667
External filtering	5295
Unsafe and *Unsafe-incomplete*	610

able operation until the next LCA change point. Note that an idle time interval may arise only when one of the assertions got verdict *Unknown* or *Unsafe*. IITL constrains idle time interval and, upon its violation, terminates current CMAV iteration, with the number of assertions on the next iteration being less at least by one. In the given case, we assume that too lengthy idle interval is due to too complicated ARG, so that it is better to rerun the algorithm for the remaining assertions. This limit determines balance of the abstract states reuse.

In addition, we suggest constraining time of any LCA interval with the help of an interval time limit (ITL). If ITL is violated, LCA will receive verdict *Unknown* and will be disabled. Too large LCA interval means that some assertion requires more refined abstraction and, most likely, will eventually violate ATL and interfere with checking of other assertions. Therefore, it is required to identify and isolate such assertions as fast as possible. However, it should be taken into account that there is a chance that this limit will not allow us to obtain verdict *Safe* or *Unsafe* for the assertion, which means that it should be used with care.

Verification facts cleaning strategies. Since we use only adjustable precision as verification facts, only adjustable precision should be cleaned at the LCA disable step. We propose different cleaning strategies depending on the place—each abstract state (ARG) or the abstract states that have not been processed yet (the so-called waitlist (WL))—from which precision is removed. Moreover, we can subtract LCA precision from the abstract state precision or completely clear abstract state precision.

In the general case, five different strategies are possible: (1) **None** (do not remove anything), (2) **WL/Subtract** (subtract LCA precision from each precision in the waitlist), (3) **ARG/Subtract** (subtract LCA precision from each precision in ARG), (4) **WL/Clear** (clear all precisions in the waitlist), and (5) **ALL** (clear all precisions in ARG).

These strategies affect the balance between positive and negative effects of the precision reuse and between time of an operation and its quality. In the general case, there exist examples in which some strategy works better than the others. All strategies have been implemented, and experiments on their comparison are presented in Section 6.

6. RESULTS OF EXPERIMENTS

In order to evaluate the suggested methods, we conducted the following experiments. Verification tasks were prepared with help of LDV Tools based on Linux kernels 3.16-rc1 and 4.0-rc1. All experiments were performed on machines with 3.4 GHz Quad Core CPU (Intel Core i7-2600), 16 Gb of RAM, and Ubuntu 12.04 (64-bit). Seventeen LDV Tools basic assertions (08_1a, 100_1a, 101_1a, 106_1a, 10_1a, 118_1a, 129_1a, 132_1a, 134_1a, 146_1a, 147_1a, 148_7a, 32_7a, 39_7a, 43_1a, 68_1a, 77_1a) [17] were used for all experiments. We compared sequential CEGAR and different variants of CMAV. For sequential CEGAR, we used CPAchecker, revision 14998, with *ldv* configuration, which includes predicate analysis [9] and explicit value analysis [10]; each launch of the CEGAR algorithm was limited to 900 seconds of CPU time and 15 Gb of RAM. For CMAV, we used LDV Tools branch *cmav* and CPAchecker branch *cmav* (revision 15941); each iteration of CMAV was limited to 1200 seconds of CPU time and 15 Gb of RAM; ATL = 900 seconds (equal to the time limit for CEGAR); IITL = 20 seconds and ITL = 100 seconds.

6.1. Adjustable Precision Cleaning Strategies

The first experiment was meant to compare different strategies of cleaning verification facts. To this end, 1000 verification tasks were prepared based on Linux kernel modules 3.16-rc1, which violate at least one assertion. Results of this experiment are presented in Table 1.

Results of experiments revealed specific features of the suggested strategies. **None** strategy shows the best results only in the case of nested error traces (i.e., when one error trace is completely contained in another), which are seldom met (about 20 times in our experiment). **ARG/Subtract** strategy is aimed at preventing negative effect of reuse of verification facts; however, the operation itself is slow (it may exhaust the iteration time limit). **WL/Subtract** strategy minimizes negative effect of precision reuse in the majority of cases and requires the least amount of resources, which allows it to find the greatest number of error

Table 4. Results of manual filtering against each assertion for the CMAV with MEA method

Assertion identifier	All Unsafes	The number of *Unsafe-incomplete*	Automated filtering	Manual filtering	New errors
08_1a	98	74 (76%)	1392	150	52
100_1a	0	0 (–)	0	0	0
101_1a	8	7 (88%)	55	10	2
106_1a	22	3 (14%)	28	26	4
10_1a	13	1 (8%)	13	13	0
118_1a	15	15 (100%)	852	26	11
129_1a	20	15 (75%)	58	35	15
132_1a	29	23 (79%)	215	48	19
134_1a	5	3 (60%)	7	6	1
146_1a	14	4 (29%)	137	21	7
147_1a	15	10 (67%)	26	24	9
148_7a	104	45 (43%)	306	152	48
32_7a	79	40 (51%)	338	130	51
39_7a	60	34 (57%)	361	116	56
43_1a	9	8 (89%)	31	15	6
68_1a	118	106 (90%)	1475	175	57
77_1a	1	1 (100%)	1	1	0
Total:	610	389 (64%)	5295	948	338

traces. **WL/Clear** strategy is better than the others only in particular cases, in which precision in ARG helps to other assertions, but precision in the waitlist (even for other assertions) does not. **ALL** strategy is closest to complete algorithm rerun (all precisions obtained for all assertions are cleared and only ARG is reused); therefore, it requires the least number of CMAV iterations. Moreover, the strategies with clearing require less iterations but lose the positive effect of verification facts reuse, while strategies with subtraction require more resources. In practice, the number of abstract states in ARG is measured by millions, whereas the waitlist contains only a few states. In comparison with basic CEGAR, all strategies are faster by about three times, since sequential CEGAR always requires 17 launches per one verification task, whereas CMAV requires only about 2 launches to solve one verification task. In the next experiments, we use **WL/Subtract** strategy as the best strategy of this experiment, since it minimizes the negative precision reuse effect in the majority of cases, keeps positive precision reuse effect, and requires reasonable amount of resources.

6.2. Verification of all Linux Kernel Modules

To compare CMAV and sequential CEGAR, all Linux kernel modules of version 4.0-rc1 were verified. The LDV Tools system prepared 6021 verification tasks for this Linux kernel version. The sequential CEGAR method worked 960 hours and had found 623 error traces.

Time. Operation time of CMAV was 200 hours, which is five times less than the operation time of sequential CEGAR. Average time spent for one iteration was 155 seconds (the limit was 1200 seconds). Maximal time of solving one verification task was 8165 seconds (time for all iterations). Analysis of experimental results revealed that time for the verification of one assertion in CMAV is less than that in CEGAR by six times because of similar actions in CEGAR in the course of checking of different assertions. However, it should be noted that, in CMAV, the time is additionally spent on auxiliary actions, for example, on processing idle intervals to launch the next iteration.

Verdicts. CMAV verdicts coincided with those in sequential CEGAR in 97.61% of cases. The number of CMAV negative transitions (*Safe* or *Unsafe* in CEGAR and *Unknown* in CMAV) is 2.17%. The number of CMAV positive transitions (*Unknown* in sequential CEGAR and *Safe* or *Unsafe* in CMAV) was 0.22%. It should be noted that there were no missed bugs (transitions from *Unsafe* to *Safe*) or additional false alarms (transitions from *Safe* to *Unsafe*) in CMAV compared to sequential CEGAR. Table 2 presents more detailed comparison of verdicts in CEGAR and CMAV.

The first cause of negative transitions in CMAV is complexity of the verification tasks created based on 17 rather than 1, like in sequential CEGAR, assertions. For example, in order to get verdict *Unsafe* in module *drivers/isdn/i4l/isdn.ko* against aspect 08_1a, sequen-

tial CEGAR requires about 830 seconds, but, in CMAV, it exhausts ATL (900 seconds). Such situations can be resolved by increasing ATL (in the given case, for example, up to 1200 seconds).

The second cause of negative transitions in CMAV is heuristic internal time limits in CMAV. For example, if we double ITL (up to 200 seconds), we find five missed verdicts *Unsafe*, with the total analysis time being increased by 1.3 times. Thus, by changing internal time limits, we change balance between verification time and quality of algorithm operation.

The main reason of positive transitions in CMAV is reusing precision and ARG between different assertions, which allows us to decrease time spent for finding error traces or proving correctness. For example, in module *fs/gfs2/gfs2.ko,* CMAV found four new error traces (*Unknown* in sequential CEGAR). In the CMAV method, this took about 700–800 seconds. With the help of sequential CEGAR, this result could be achieved if we increased time limit up to 1400–1500 seconds. Thus, CMAV solves verification tasks that cannot be solved in basic CEGAR with the same resource limitations.

Iterations. The total number of CMAV iterations was 8395; i.e., the ratio of the iterations to the verification tasks was equal to $1=1.39$. CMAV solves 5511 verification tasks (almost 92%) in the course of the first iteration (i.e., only 8% of all tasks require extension from MAV to CMAV). The maximal number of iterations for one verification task was 15 (all assertion verdicts were *Unknown;* some of them were obtained in the course of one iteration).

6.3. Comparison of CMAV with MEA and MEA

To evaluate CMAV with MEA, we used verification tasks obtained on the basis Linux kernel modules of version 4.0-rc1 that violate at least one of the assertions being verified (i.e., 624 verdicts *Unsafe* from the previous experiment). The main goal of this experiment was to compare results obtained by methods CMAV with MEA and MEA.

Comparison with CMAV. Compared to CMAV, the operation time increased from 83 hours to 155 hours (i.e., by almost two times, which is similar to comparison of basic CEGAR with MEA). The number of CMAV iterations increased from 1039 to 1172. Fourteen verdicts *Unsafe* were lost (were changed to *Safe*) because of nested error traces; 389 verdicts *Unsafe* out of the remaining 610 verdicts (i.e., 64%, which is similar to MEA) turned to *Unsafe-incomplete*. The basic cause of missing verdicts *Unsafe* is that some assertions are mutually related; therefore, the method believes that there is only one error. In the MAV method, this problem is solved on the *LCA disable* step (Fig. 3); however, in the MAV with MEA algorithm this step is omitted (Fig. 5). This problem can be resolved by modifying the assertion in such a way that it could be ignored; however, this will result in further complication of verification tasks. An attempt to implement this idea resulted in loss of about one hundred of other verdicts *Unsafe*.

Automated filtering. For 610 verdict *Unsafe* and *Unsafe-incomplete* the CMAV with MEA method found more than 500 thousand error traces; therefore, this method cannot be efficiently used without further filtering. Results of automated filtering of error traces found are presented in Table 3. For internal filtering, the ABE filter [9] of static tool CPAchecker was used, which made it possible to waive about 94% error traces "on the fly," i.e., in the course of the algorithm operation with insignificant overheads. External filtering (based on filter of assertion model functions of the LDV Tools system) keeps less than 1% of all error traces; the total time of operation of this filter is four hours. Thus, 99% of all error traces obtained can be filtered automatically. The 5295 error traces obtained in this case can be considered to be the final result of the CMAV with MEA method, since the remaining error traces can help the user in one way or another to understand how to fix one or another bug.

Manual filtering. In order to obtain the number of new errors more accurately, manual filtering is needed. Manual filtering was performed in the way that is used in the framework of the MEA method (Section 3.5). It took about four days by one person. Manual filtering results against each assertion are presented in Table 4.

For some assertions (for example, 08_1a and 118_1a), a great number of the so-called repeated errors are found (i.e., one error can be associated with an infinite number of traces). For instance, this may happen if an error is in function *probe*, which may be called an arbitrary number of times. Such cases, as a rule, contain many error traces (the maximal number of error traces corresponding to one repeated error was 528; altogether, 3476 error traces were associated with repeated errors), and these cases result in very high percentage (up to 100%) of verdicts *Unsafe-incomplete*. However, they are easily resolved since they require only one action upon manual filtering. In such cases, there is no need to try to find "all" error traces. At the same time, for some assertions (for example, 148_7a, 32_7a, and 39_7a), verdicts *Unsafe* are met much more frequently. For these verdicts, there are few error traces that correspond to new errors. The maximum number of new errors for a verification task was seven (for assertion 08_1a).

Conclusions. Thus, the total number of errors found by the CMAV with MEA method was 948 (including 338 new errors), which is 1.5 times more than that for the CMAV method. Note that this result coincides with the results of using MEA. More than 99% of error traces found by this method are processed automatically. It should be emphasized that the CMAV with MEA method allows us to find all errors

for almost 40% of verification tasks. We note also that, in the majority of cases, all errors cannot be found just because of the repeated ones. This method requires twice as much resource as CMAV does, as well as considerable additional time for manual filtering. The experiment described demonstrates that CMA with MEA shows the same results as MEA.

7. DIRECTIONS OF FUTURE STUDIES

We plan to carry out more experiments with the proposed method in order to determine its optimal parameters, such as interval limits, strategies of verification facts reuse, algorithms for automated filtering of error traces, etc. This will allow us to develop recommendations for users, which will help them to choose algorithm configuration suitable for a particular task.

Besides, it can be combined with the bug finding method. It often happens that a program cannot be completely analyzed only because of a small fragment (for example, a deep loop). Then, the internal limits could help to determine fragments of the program that are too difficult for verification in order to miss their analysis. In this case, it is impossible to prove correctness of the program, but one can find more errors.

One of the basic problems of MAV is more complicated verification tasks obtained upon instrumentation. Code instrumentation adds additional verifications describing assertions, which makes the original code more complicated. In the case of MAV, the number of such additional verifications is great (several thousand lines of the code), which increases verification time. This difficulty can be overcome by adopting alternative variants of creation of verification tasks, for example, by describing assertions in a special language accepted by the static verifier. Besides, this may help to resolve the problem of lost verdicts *Unsafe* in the CMAV with MAE method due to nested error traces.

However, the number of assertions in practical tasks is measured by hundreds, and the overheads associated with processing them can cancel all benefits of the proposed method. The way out of this in the given case is to develop a strategy for partitioning assertions to be verified into smaller groups of assertions for more efficient verification (for example, by minimizing intersections between the assertions in a group in terms of some criterion). Such groups of assertions can further be verified by MAV in parallel to improve efficiency of the resource (processor time) use and speed up the calculations.

8. CONCLUSIONS

The results of conducted experiments confirm, that the suggested static verification methods CMAV and CMAV with MEA are promising. The CMAV method reduces total time of verification in five times in comparison with basic CEGAR with minor loss of results. The CMAV with MEA method is capable of finding all errors for all requirements being verified.

Naturally, it requires more time for verification in 1.2 times in comparison with the CMAV method, and manual filtering of error traces. The CMAV with MEA method finds all possible errors for all verified assertions for almost 40% of the problems, with the number of new errors found being 1.5 times greater than that in the CMAV method.

The CMAV with MEA method operates five time faster in comparison with the method MEA, and provides similar results.

Therefore, the CMAV method is more aimed at proving correctness of the program against several requirements, in case of finding any violations it requires their iterative fixing. The CMAV with MEA method is aimed at finding all errors of all requirements being verified, but it requires more resources in comparison with the CMAV method.

ACKNOWLEDGMENTS

This work was supported by the Ministry of Education and Science of the Russian Federation (unique project identifier RFMEFI61614X0015).

REFERENCES

1. Beyer, D., Competition on software verification, *Lect. Notes Comput. Sci.,* 2012, vol. 7214, pp. 504–524. http://dx.doi.org/. doi 10.1007/978-3-642-28756-5_38
2. Beyer, D., Software verification and verifiable witnesses, *Lect. Notes Comput. Sci.,* 2015, vol. 9035, pp. 401–416. http://dx.doi.org/. doi 10.1007/978-3-662-46681-0_31
3. Khoroshilov, A., Mutilin, V., Novikov, E., et al., Towards an open framework for C verification tools benchmarking, *Lect. Notes Comput. Sci.,* 2012, vol. 7162, pp. 179–192. http://dx.doi.org/. doi 10.1007/978-3-642-29709-0_17
4. List of errors revealed in Linux Kernel modules by means of the LDV Tools system. http://linuxtesting.org/results/ldv.
5. Shved, P., Mandrykin, M., and Mutilin, V., Predicate analysis with BLAST 2.7, *Lect. Notes Comput. Sci.,* 2012, vol. 7214, pp. 525–527. http://dx.doi.org/. doi 10.1007/978-3-642-28756-5_39
6. Beyer, D., Henzinger, T.A., Jhala, R., and Majumdar,R., The software model checker BLAST, *Int. J. Software Tools Technology Transfer,* 2007, vol. 9, nos. 5–6, pp. 505–525. http://dx.doi.org/. doi 10.1007/s10009-007-0044-z
7. Beyer, D. and Keremoglu, M.E., Cpachecker: A tool for configurable software verification // computer aided verification, *Lect. Notes Comput. Sci.,* 2011, vol. 6806, pp. 184–190. http://dx.doi.org/ doi 10.1007/978-3-642-22110-1_16.10.1007/978-3-642-22110-1_16
8. Beyer, D. and Wendler, P., Reuse of verification results: Conditional model checking, precision reuse, and ver-

ification witnesses, *Lect. Notes Comput. Sci.,* 2013, vol. 7976, pp. 1–17. http://sv-comp.sosy-lab.org/.

9. Beyer, D., Keremoglu, M.E., and Wendler, P., Predicate abstraction with adjustable-block encoding, in *Formal Methods in Computer-Aided Design,* 2010.
10. Beyer, D. and Löwe, S., Explicit-state software model checking based on CEGAR and interpolation, *Lect. Notes Comput. Sci.,* 2013, vol. 7793, pp. 146–162.
11. Clarke, E.M., Grumberg, O., Jha, S., et al., Counterexample-guided abstraction refinement, *Lect. Notes Comput. Sci.,* 2000, vol. 185 5, pp. 154–169.
12. Mandrykin, M.U., Mutilin, V.S., and Khoroshilov, A.V., Introduction to CEGAR: Counterexample-guided abstraction refinement, *Tr. Inst. Sistemnogo Program. Ross. Akad. Nauk,* 2013, vol. 24, pp. 219–292.
13. Novikov, E.M., An approach to implementation of aspect-oriented programming for C, *Program. Comput. Software,* 2013, vol. 39, no. 4, pp. 194–206. http://dx.doi.org/. doi 10.1134/S0361768813040051
14. Beyer, D., Löwe, S., Novikov, E., et al., Precision reuse for efficient regression verification, *Proc. of the 9th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on Foundations of Software Engineering,* St. Petersburg, Russia, 2013.
15. Beyer, D., Henzinger, T.A., Keremoglu, M.E., and Wendler, P., A technique to Pass Information between Verifiers, *Proc. of the 20th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering,* Cary, NC, 2012.
16. Mordan, V. and Novikov, E., Minimizing the number of static verifier traces to reduce time for finding bugs in linux kernel modules, *Proc. of the Spring/Summer Young Researchers' Colloquium on Software Engineering,* 2014, vol. 8.
17. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., et al., Configurable toolset for static verification of operating systems kernel modules, *Program. Comput. Software,* 2015, vol. 41, no. 1, pp. 49–64.

Translated by Alexander Pesterev