Checking Several Requirements at once by CEGAR*

Vitaly Mordan and Vadim Mutilin

Institute for System Programming of the Russian Academy of Sciences
{mordan,mutilin}@ispras.ru

Abstract. At present static verifiers, which are 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. If we use this approach for checking several requirements at once, then finding a first violation of some requirement or exhausting resources for some requirement will prevent checking the program against other requirements. In particular we may miss violations of some requirements. That is why in practice each requirement to the program is usually checked separately. However, static verifiers perform similar actions during checking of the same program against different requirements and thus a lot of resources are wasted. This paper presents a new CEGAR-based method for software static verification, that is aimed at checking programs against several requirements at once and getting the same result as basic CEGAR, which checks requirements one by one. In order to do it the suggested method divides resources among requirements equally and continues analysis after finding violation of requirement excluding that requirement. We used Linux kernel modules to conduct experiments, in which implementation of the suggested method reduced total verification time by 5 times. The total number of divergent results in comparison with CEGAR was about 2%.

Keywords: static verification, counterexample guided abstraction refinement, aspect.

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 finding frequent bugs. The main disadvantage, which makes it much less applicable in practice, is a large amount of required resources such as CPU time and memory especially for large software systems. Currently static verification tools (static verifiers), based on Counterexample Guided Abstraction Refinement

* The research was carried out with funding from the Ministry of Education and Science of Russia (the project unique identifier is RFMEFI61614X0015).

(CEGAR), are appropriate for large software systems which is demonstrated in annual Competitions on Software Verification [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 static verifiers check the program against all specified requirements. The CEGAR approach checks the program against only one requirement at a time, because it stops after finding a bug (and thus other requirements will not be checked) and checking the program against some requirement may exhaust all given resources. Thus large amount of required resources for verification further increases depending on the number of requirements.

Let us consider an example which demonstrates this problem. Linux Driver Verification Tools (LDV Tools) are an open source toolset for checking correctness of Linux kernel modules against rule specifications (i.e. specifications of rules for correct usage of the kernel API) with help of different static verifiers [3]. It has already helped to find more than 190 bugs in Linux kernel modules [4]. The process of verification of all Linux kernel modules with help of LDV Tools against a single rule specification by static verifiers BLAST [5, 6] or CPAchecker [7] takes about 2 days, whereas verification of 17 basic rule specifications takes about 40 days. But the number of rule specifications is 50 and a few new rule specifications are under development.

This paper presents a new static verification method which can significantly reduce required time even if the number of requirements to be checked is growing. 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 as well as CEGAR do, but faster. This method was implemented as extensions of LDV Tools [3] and the CPAchecker [7] static verifier and was evaluated on verification of Linux kernel modules.

We make the following contributions:

- We propose a new method for static verification which extends the CEGAR approach for checking a program against several requirements at once.
- We implement this method as extensions of LDV Tools and the CPAchecker static verifier.
- We experimentally show that the suggested method reduces total verification time by 5 times on verification of Linux kernel modules, the total number of negative results, in which the suggested method requires more resources than CEGAR in order to prove correctness of a program against specified requirement or to find its violation, is about 2%.

Next Section describes the CEGAR approach and introduces definitions which are used in the paper. Section 3 presents related work. In Section 4 the new method is suggested. Section 5 presents its implementation. In Section 6 the results of experiments are described.

2 Background

2.1 Definitions

In the paper we refer to an **aspect** as a formal representation of a checked requirement to a program [8]. Aspects represent what we intend to check in the program. For example: *allocated resources should be correctly freed*. Indeed, specifications of rules for correct usage of the kernel API are aspects.

Error location is a location in a program, which corresponds to an aspect violation (for example, predefined function *error*() or predefined label *ERROR*).

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

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

Verdict is a result of solving a verification task by a static verifier. Usually there are 3 possible verdicts:

- *Safe*: a program is correct against a specified aspect;
- *Unsafe*: a program violates a specified aspect, a corresponding violation is represented as an error trace;
- *Unknown*: a static verifier cannot solve a given verification task.

2.2 Counterexample Guided Abstraction Refinement

Since it is usually impossible to analyze a precise model of a program in a reasonable time, CEGAR operates with an abstract model of the program.

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 a program [6].

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

Verification fact is a result (possibly intermediate) of the verification process, that is necessary for proving correctness of the program against a given aspect [9].

Abstraction precision (precision) is a verification fact, that instructs the analysis which information should be tracked and which information should be omitted in abstraction of the program [9]. For example, in the predicate analysis [10] precisions define tracking predicates, in the explicit value analysis [11] precisions define tracking variables. Thus a precision defines the current level of abstraction in ARG.

The CEGAR algorithm is presented in Fig. 1 [12, 13]. At the beginning 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 a found counterexample for feasibility. If it is feasible then an error trace is built for the found 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 general case, static verifiers, which implement the CEGAR approach, operate with limited resources (such as CPU time and memory) in practice and terminate their analysis with the verdict *Unknown* in case of resources exhaustion.

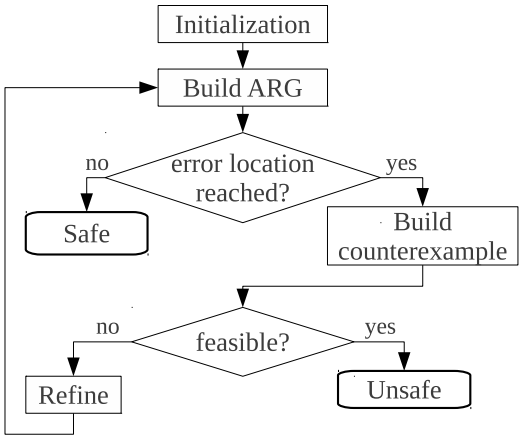

Fig. 1. The CEGAR loop.

2.3 Example

Let us consider an example. There are three aspects:

Aspect 1 : *all allocated resources by usb_alloc_urb() should be freed by usb_free_urb().*
Aspect 2 : *the same mutexes should not be acquired or released twice in the same process.*
Aspect 3 : *an offset should not be greater than a size of an array.*

In order to get verdicts for all specified aspects with the basic CEGAR approach we need to prepare 3 different verification tasks and run the CEGAR algorithm 3 times. Each verification task is created based on program source code and a specified aspect. In a verification task a selected aspect corresponds to specific error location. For example, **Aspect_i** can correspond to error label *ERROR_i*, i=1,2,3. Thus **Aspect_3** may be represented by the following checks in the verification task:

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

3 Related work

The idea of modifying the CEGAR algorithm in order to reduce verification time is not new. By adding auxiliary actions at different steps of the CEGAR algorithm (Fig. 1) it is possible to solve specific verification tasks faster.

3.1 Regression verification

Even if a program is absolutely correct, it always can be modified. Every modification of the program potentially may add new bugs. Such bugs are called regressions. Regression verification is aimed at verifying program revisions in order to find regressions. One of the approaches to do it efficiently is to reuse verification results [9]. Verification facts, which were obtained during analysis, are stored as results and then are used on the *Initialization* step of the CEGAR algorithm (Fig. 1) to start analysis with the known level of abstraction, for example, to verify the next revision of that program. Experiments confirm [9], that reuse of verification facts reduces time for regression verification. At the same time in some cases reuse of verification results may increase time of analysis. For example, abstraction may become too accurate for the new revision (for some verification tasks precision reuse increased time almost twice [9]). Regression verification approaches are useful for reducing time for verification, but it is unclear whether verification facts can be reused between different aspects as well as between program revisions or not.

3.2 Conditional model checking

The verdict *Unknown* means that a static verifier fails to solve a verification task. It is still unclear whether the given program correct or not and thus resources were spent for nothing. In order to solve this problem the conditional model checking approach was suggested [14]. A static verifier saves its result even if it cannot solve the whole verification task. This result describes which parts of the program (e.g., abstract states in ARG) were successfully verified and which were not. Then another static verifier (or the same with another configuration) takes this result at the *Initialization* step of the CEGAR algorithm (Fig. 1) and verifies only those parts of the program which were not verified. Conditional model checking helps to solve problems which cannot be solved by a single launch of a static verifier [14]. Conditional model checking can be useful for resolving verdicts *Unknown*.

3.3 Method for finding all violations of an aspect

Any program can contain any number of bugs for a given aspect. The basic CEGAR algorithm stops after it finds a first bug. In practice it significantly increases time for finding and fixing different bugs of the same kind. In order to solve this problem the method for finding all violations of an aspect was

suggested [15]. Its main idea is to continue analysis after finding a bug. Obviously, this method requires more time for analysis (in comparison with the basic CEGAR), but it also reduces time for finding and fixing different bugs of the same kind. Nevertheless, analysis may be terminated abnormally after finding several bugs, for example, it may exhaust resources that means that there may be more bugs. That is why this method cannot be used directly for checking several aspects at once.

4 Multi-Aspect Verification

Multi-Aspect Verification (**MAV**) is aimed at configurable checking of several aspects for a given verification task at once. 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 aspects (for more details see Section 5). The main requirement to MAV is to get the same result for all aspects (i.e. prove correctness or find error traces) as basic CEGAR for each aspect, but faster.

The first problem that must be resolved, is that basic CEGAR checks for a single error location, which corresponds to a single aspect. It is possible to check for several error locations (for example, check for several error labels at once), but those error locations do not correspond to specific aspects. Thus, if we intend to check for several error locations, each of them should get link to a corresponding aspect (for example, to determine which aspect is violated in a found error trace).

Second, checking of some aspect may interfere with checking of another one. The basic CEGAR approach stops after it finds a first error trace, thus violation of a single aspect leads to analysis termination for all aspects. Also it is impossible to limit resources for some aspect separately, thus if CEGAR cannot check a single aspect (for example, it exhausts time limit), then analysis of all aspects will be terminated.

So, the MAV method should satisfy the following requirements:

- to differ one aspect from another (in terms of error locations);
- to continue analysis after finding an error trace (the verdict *Unsafe*);
- to identify the verdicts *Unknown* and to continue analysis after them;
- to get the same verdicts for aspects as basic CEGAR;
- to consume less resources (in comparison with basic CEGAR).

4.1 The Multi-Aspect Verification method

The MAV algorithm was designed to meet the mentioned above requirements (Fig. 2). It extends the basic CEGAR algorithm and checks for several error locations.

In terms of MAV each **aspect** contains the following attributes: an unique identifier, a corresponding error location identifier (for example, specific error

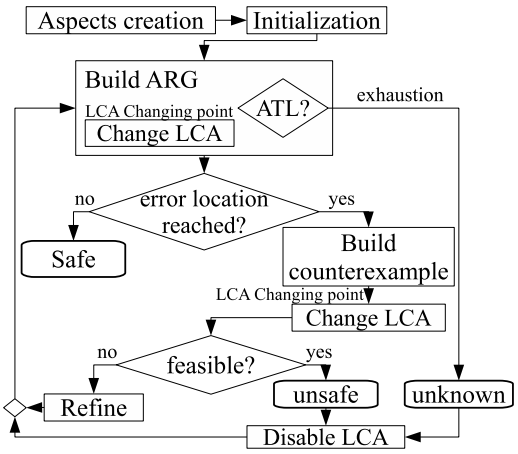

Fig. 2. The MAV algorithm.

label name), an aspect verdict, consumed resources (CPU time, wall time, etc.), corresponding verification facts. At the *Aspects creation* step (Fig. 2) links between aspects and error locations are created, attributes for each aspect get initial values.

Aspect verdict is an internal verdict of an aspect. Aspect verdict takes all common verdict values (which were defined in Section 2) and auxiliary value *Checking*, which means that this aspect is currently being checked.

In order to know, which aspect is being checked (to change aspect verdicts, to keep records of consumed resources, etc.), the notion of **Latest Checked Aspect** (**LCA**) is suggested. LCA is the latest aspect, which was checked during analysis. But during the ARG construction more than one aspect can be checked at once (i.e. ARG can be built based on several aspects at once). The following approximation is suggested: *only one aspect is being checked at the moment of time and the latest checked aspect is LCA*.

At the start of the algorithm LCA is unset. Then all time of analysis can be divided by **LCA changing points**, which are moments of time, when LCA changes. Current checked aspect is considered equal to LCA until the next LCA changing point. The main idea of these points is that they should represent that the algorithm builds ARG for the selected aspect. Since counterexample always contains reached error location, it is always possible to determine which aspect it violates and to add LCA changing point after the *Build counterexample* step. In general case LCA changing points can be added at the *Build ARG* step.

After that it is possible to divide an analysis time line for the **LCA intervals**, which are time intervals between two LCA changing points. Thus we can use the LCA intervals to calculate time, consumed by each aspect. In order to do it we add time of the interval to LCA time at the *Change LCA* step.

In general case verification facts for each aspect are obtained after the *Change LCA* step for LCA. For example, if verification facts are represented as precisions,

we can obtain them for every infeasible counterexample after the *Refine* step. They define a current level of abstraction for LCA.

Tracking aspect time gives us the possibility for **Aspect Time Limit** (**ATL**) for each aspect. ATL is aimed at limiting total time for aspect. In case of ATL exhaustion, LCA will be disabled on the *Disable LCA* step (Fig. 2). Since in general case the LCA interval can exceed ATL by itself, ATL should be checked asynchronously inside the LCA interval (i.e. during the *Build ARG* step).

Therefore aspect verdicts are changed during analysis in the following way. At the *Aspects creation* step every aspect gets the aspect verdict *Checking*. In case of finding an error trace for some aspect, its aspect verdict changes to *Unsafe*. In case of ATL exhaustion, the corresponding aspect verdict changes to *Unknown*. If the algorithm has finished with the verdict *Safe*, then all aspects with the aspect verdict *Checking* change to *Safe*.

The *Disable LCA* step consists of the following operations: (1) stop checking error location, which corresponds to LCA, (2) remove verification facts relevant to LCA and (3) unset LCA. Verification facts, which correspond to some aspect, could both be useful for the others and interfere with the others (for more details see Section 6). It is possible to remove abstract states, relevant to LCA, from ARG as well, but we should take into account that any operation with full ARG may not be efficient (since ARG can consist of millions abstract states). Therefore we need options to determine, which verification facts we intend to track and where we intend to remove them.

The *Disable LCA* step unsets LCA, thus, for the next interval we do not have LCA until the next LCA changing point. We call such interval the **Interval without LCA** (except the first one). In terms of current approximation such intervals means that analysis does not check any aspect and thus time is wasted. It leads to extending the idea of ATL for the notion of **Internal Time Limit** (**ITL**). The main idea of any ITL is to limit time for some operation (or sequence of operations) and then execute predetermined action in case of its exhaustion. Thus ATL is ITL, which limits total time for the aspect and in case of its exhaustion that aspect will be disabled. Potentially we can also limit the LCA interval, the first interval, the interval without LCA, etc.

The suggested approximation can be extended for more general case, in which more than one LCA can be at a time. In that case the notion of **Set of Latest Checked Aspects** (**SLCA**) is suggested. This approximation is more close to reality (since MAV may actually check for several aspects at a time), but cannot solve problem of tracking time and verification facts for each aspect as clearly as LCA approximation.

4.2 The Conditional Multi-Aspect Verification method

The main disadvantage of MAV is that it cannot continue analysis if the algorithm was somehow terminated. ATL helps to find time limit exhaustion for specific aspect, but memory limit exhaustion and abnormal termination of static verifiers are still remain as unresolved problems. In such cases all aspects with the aspect verdict *Checking* get the aspect verdict *Unknown*. But if only one aspect

exceeds memory limit we could expect that MAV isolates it. In order to achieve this, MAV was extended based on ideas of Conditional model checking [14]. The main idea is to launch the MAV method a few times, passing intermediate results between each launch, until all aspects get verdicts. We call this extension **Conditional MAV** (**CMAV**). The schema of CMAV is presented in Fig. 3.

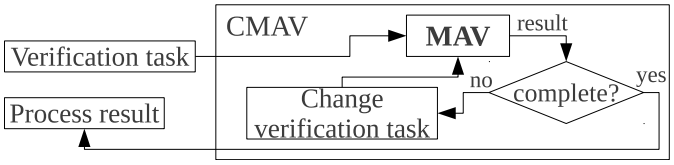

Fig. 3. The CMAV schema.

In order to save all actual information about verification process we suggest common format, which keeps aspect attributes (an aspect identifier, an aspect verdict, a consumed time, an error trace identifier, a reason of *Unknown*) and LCA identifier. This intermediate result in common format is stored into the specified file. At the *Change LCA* step, the *Disable LCA* step and analysis termination information in this file is updated. Thus in case of any abnormal termination this file will contain all relevant information, including LCA, which might cause the termination.

We call each launch of the MAV algorithm an **iteration** of CMAV. After each iteration CMAV determines if analysis have been completed based on the file with intermediate results. If the algorithm was terminated abnormally, CMAV finds the reason of its termination and the aspect (or aspects), which caused it. In case of global problems (such as incorrect verification task), all aspects get the aspect verdict *Unknown* and CMAV terminates its analysis. Otherwise only those aspects which caused termination get the aspect verdict *Unknown* (for example, in case of memory limit exhaustion, while all aspect verdicts are still *Checking*, LCA gets the aspect verdict *Unknown*). If analysis is not completed, CMAV starts new iteration, which checks aspects with the aspect verdict *Checking*. In any case next iteration gets at least one less aspect to check. Thus, number of all iterations is less or equal than the number of all aspects. After the completion of the last iteration CMAV unites intermediate information from all iterations and presents it as the final result for the given verification task.

Also CMAV helps to resolve problem with the long intervals without LCA, the main reason of which is too complex ARG. Such intervals are possible only after the *Disable LCA* step, i.e. at least one aspect have already got the aspect verdict *Unsafe* or *Unknown*. In order to limit such intervals we suggest **Idle Interval Time Limit** (**IITL**). IITL is ITL, which limits the interval without LCA. In case of IITL exhaustion CMAV starts new iteration, which gets at least one aspect less to check, since at least one aspect got its aspect verdict on the previous iteration (before the *Disable LCA* step).

5 Implementation

We have implemented the suggested method as extensions of LDV Tools and the CPAchecker static verifier. LDV Tools prepare verification tasks, based on multiple aspects, and launch CMAV iterations with processing results in the CMAV common format. The CPAchecker static verifier executes the MAV algorithm.

5.1 LDV Tools extension

LDV Tools take multiple aspects as input and join them into a Combined aspect. The Combined aspect contains references to the original aspects, for example, each error location corresponds to some original aspect. For the preparation of the Combined aspect LDV Tools were equipped with the new component merging aspects.

The CPAchecker input configuration includes references to original aspects. Therefore we can choose the set of original aspects to check while passing verification task prepared with help of the Combined aspect.

The wrapper inside LDV Tools which launches CPAchecker reads CPAchecker output in the CMAV common format to determine whether analysis is completed or the next CMAV iteration is needed. In the later case the wrapper changes verification task by removing references to corresponding aspects from the CPAchecker input configuration. Thus on the next iteration CPAchecker ignores original aspects for which we have already got verdicts.

If the analysis is completed then LDV Tools prepare final report containing information for every original aspect (its verdict, consumed time, error trace, reason of *Unknown*), so it can be easily compared with CEGAR final reports.

5.2 CPAchecker with MAV

We extended the CPAchecker static verifier in order to support the MAV method based on its description in Section 4. Here implementation details are presented.

Aspect. In CPAchecker the notion of error location is extended for the notion of property, which is represented by automaton. In automaton there is a state (or states) specified as an error state and the task is to prove that the error state cannot be reached. LDV Tools pass to CPAchecker such automata with references to the original aspects with verification task. Such references represent different error labels in program source code, thus automata check that corresponding error labels cannot be reached.

In order to check a few properties at once we suggest automaton composition. In automaton composition every automaton gets a unique name and can be disabled. The *Aspects creation* step creates aspects for every passed automaton, aspects identifiers are equal to corresponding automaton names. The *Disable LCA* step disables corresponding automaton.

In general case every error location can correspond to some part of the automaton (for example, each error location corresponds to some automaton transition). Then such automaton parts should get unique names at the *Aspects creation* step and can be disabled at the *Disable LCA* step.

LCA. We implemented approximation, in which only one aspect is being checked at a time. LCA changing points happen only after the *Build counterexample* step.

Adjustable precision. We extended abstraction precision based on set theory. At the start precision is empty, which corresponds to an empty set. We added operations of addition and subtraction between two precision and operation of clearing precision. We called such extended precision as adjustable precision. Thus every used type of precision in analysis are adjustable precision.

Furthermore only adjustable precision is used as verification fact. At the *Aspects creation* step each aspect gets empty precision as corresponding verification fact. After finding infeasible counterexample it is possible to get new precision (from the *Refine* step) and add it to precision of LCA.

ITLs. In addition to suggested above ITLs (ATL for MAV and IITL for CMAV) we implemented 2 heuristic ITLs. **Basic Interval Time Limit** (**BITL**) limits each LCA interval, in case of its exhaustion LCA is disabled with the aspect verdict *Unknown*. **First Interval Time Limit** (**FITL**) limits the first interval, in case of its exhaustion CPAchecker terminates its analysis, all *Checking* aspects get the aspect verdict *Unknown*. The main reason of additional ITLs is to find the aspect verdict *Unknown* faster (since it means that we neither could prove aspect correctness nor find its violation). Any of those ITLs can be unset.

Cleaning adjustable precision. Since we use only adjustable precision as verification facts, only adjustable precision should be cleaned at the *Disable LCA* step. We can remove corresponding adjustable precision from every abstract state (the whole ARG) or only from not yet processed abstract states (so called waitlist). Moreover, we can subtract precision, which corresponds to LCA, or clear it. In general case 5 different strategies are possible: (1) **None** (do not remove anything); (2) **WL/Subtract** (subtract from each precision in waitlist precision, that correspond to LCA); (3) **ARG/Subtract** (subtract from each precision in ARG precision, that correspond to LCA); (4) **WL/Clear** (clear all precisions in waitlist) and (5) **ALL** (clear all precisions in ARG).

The suggested strategies represent different techniques to precision reuse effect between different aspects (up from full precision reuse in **None** strategy to not using this effect at all in **All** strategy). In general case there are examples, in which some strategy is better, than the others. All strategies were implemented, experiments with their comparison are presented in Section 6.

6 Results of the experiments

In order to evaluate the suggested method 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 3.4GHz Quad Core CPU (Intel Core i7-2600), 16Gb of RAM and Ubuntu 12.04 (64-bit). 17 LDV Tools basic aspects [16] were used for all experiments. For basic CEGAR we used CPAchecker revision 14998 with *ldv* configuration, which

includes predicate analysis [10] and explicit value analysis [11], 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 (same as time limit for CEGAR), IITL=20 seconds, BITL=100 seconds, FITL=100 seconds.

6.1 Cleaning adjustable precision strategies

The first experiment is meant to compare different strategies of cleaning adjustable precision. In order to do it 1000 verification tasks were prepared based on Linux kernel modules 3.16-rc1, which violate at least one aspect. The results of this experiment are presented in Table 1.

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

Table 1. Cleaning adjustable precision strategies comparison (l is an average number of iterations/launches for one verification task).

Experiment results revealed the features of the suggested strategies. **None** strategy shows the best result only in case of nested error traces (i.e. there are two error traces for different aspects and the first one is contained in the second), but their numbers are low (about 20 in the experiment). **ARG/Subtract** is aimed at preventing negative precision reuse effect at all, but the operation itself is slow (it may exhaust the iteration time limit). **WL/Subtract** strategy minimizes such negative precision reuse effect for most cases and thus finds the maximal number of error traces. **WL/Clear** strategy is better than the others only in particular cases, in which precision in ARG helps to other aspects, but precision in waitlist (even for other aspects) does not. **ALL** strategy is unexpectedly fast (it also requires the lowest number of CMAV iterations) since it fully removes negative precision reuse effect with minimal number of additional operations. At the same time **ALL** strategy loses so called positive precision reuse effect, which allows to solve verification tasks that were not solvable in CEGAR (that is demonstrated in the next experiment). In comparison with basic CEGAR all strategies are faster in about 3 times, since 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 most cases, keeps positive precision reuse effect and requires reasonable resources.

6.2 Verification of all Linux kernel modules

In order to evaluate the suggested method all Linux kernel modules of version 4.0-rc1 were verified with the help of CEGAR and CMAV. LDV Tools prepare 6021 verification tasks for that Linux kernel version. The basic CEGAR approach total time is 1047 hours (CPU time), LDV Tools total time (together with CEGAR) is about 960 hours (wall time), the total number of found error traces is 623.

Time. CPAchecker with CMAV total time is 360 hours (CPU time), which is 3 times less than for CEGAR, LDV Tools total time (together with CMAV) is 200 hours (wall time), which is 5 times less than with CEGAR. Average time for one iteration is 155 seconds (it is limited to 1200 seconds). Maximal time for a verification task is 8165 seconds (time of all iterations). Analysis of experimental results revealed, that aspect time in CMAV (which is total LCA intervals time) is less than corresponding time in CEGAR on average at 6 times, because of similar actions in CEGAR during checking of different aspects. But we should take into account, that CMAV contains the intervals without LCA for performing common or auxiliary actions, so the full CPU time of CMAV is 3 times less than for CEGAR.

Verdicts. CMAV verdicts are the same as in CEGAR in 97.61%. 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 CEGAR and *Safe* or *Unsafe* in CMAV) is 0.22%. There are no missed bugs (transitions from *Unsafe* to *Safe*) or additional false alarms (transitions from *Safe* to *Unsafe*) in CMAV in comparison with CEGAR. Table 2 contains more detailed comparison of verdicts in CEGAR and CMAV.

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

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

The first reason of negative transitions in CMAV is more complex verification tasks, which may require more time. For example, in order to find the verdict *Unsafe* in module *drivers/isdn/i4l/isdn.ko* against aspect 08_1a CEGAR requires about 830 seconds, but in CMAV it exhausts ATL (900 seconds). They still can be found if we increase ATL (in case of *drivers/isdn/i4l/isdn.ko* for about 1200 seconds). The second reason of negative transitions in CMAV is heuristic ITLs in CMAV. For example, up to 48% of the lost verdicts *Safe* for each aspect could

be found in case of unsetting FITL. At the same time, total time of analysis also increases in almost 1.3 times. Thus by changing ITLs it is possible to change balance between verification time and possible inaccurate results.

The main reason of positive transitions in CMAV is reusing precision and ARG between different aspects, which may decrease time for finding error traces or proving correctness. For example, in module *fs/gfs2/gfs2.ko* CMAV could find 4 new error traces, which are *Unknown* in CEGAR. In the CMAV method corresponding error traces are found for about 700-800 seconds. The CEGAR algorithm also can find them, but for about 1400-1500 seconds. Thus CMAV solves verification tasks, that cannot be resolved in basic CEGAR with the same resource limitation.

Iterations. Total number of CMAV iterations is 8395, and thus the ration between iterations and verification tasks is l=1.39. CMAV solves 5511 verification tasks (almost 92%) by the first iteration (i.e. for 92% of all tasks the MAV method is enough, 8% needs the CMAV method). Maximal number of iterations for one verification task is 15 (all aspect verdicts are *Unknown*).

7 Conclusion

The suggested method extends the CEGAR approach and provides a means to check several requirements at once. Experiments showed that its implementation in the CPAchecker static verifier and LDV Tools worked 5 times faster than LDV Tools with the CEGAR approach, CPAchecker time was reduced by 3 times. More than 90% of all verification tasks require only one launch of the CPAchecker static verifier to check program against all requirements. At the same time verdicts for verification tasks are the same for about 97.61% of all verification tasks. Some verification tasks (about 2.17%) need more resources than in basic CEGAR. At the same time positive effect of precision reuse helps to solve some tasks (about 0.22%), that cannot be resolved in basic CEGAR with the same resource limitation, and even the total number of found error traces in CMAV is greater than in CEGAR.

The main benefit of the suggested method is more optimal usage of resources. Also it can be configured by using different options (for example, internal time limits) for specific verification task and user demands, it is possible to determine balance between verification time and quality of analysis.

The suggested method was presented and implemented as the CEGAR extension, but potentially the same ideas could be used for the other static verification approaches. One of the perspective area of the future development is integration of the suggested method with the method for finding all violations of an aspect, which will allow to find all violations of several aspects at once.

References

1. Beyer, D.: Competition on software verification. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 7214 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2012) 504–524

2. Beyer, D.: Software verification and verifiable witnesses. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 9035 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2015) 401–416
3. Khoroshilov, A., Mutilin, V., Novikov, E., Shved, P., Strakh, A.: Towards an open framework for C verification tools benchmarking. In: Perspectives of Systems Informatics. Volume 7162 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 179–192
4. Web-site: Problems found in Linux kernels. http://linuxtesting.org/results/ldv
5. Shved, P., Mandrykin, M., Mutilin, V.: Predicate analysis with BLAST 2.7. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 7214 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 525–527
6. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: The software model checker BLAST. International Journal on Software Tools for Technology Transfer **9**(5-6) (2007) 505–525
7. Beyer, D., Keremoglu, M.: CPAchecker: A tool for configurable software verification. In: Computer Aided Verification. Volume 6806 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 184–190
8. Novikov, E.: An approach to implementation of aspect-oriented programming for C. Programming and Computer Software **39**(4) (2013) 194–206
9. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE 2013, St. Petersburg, Russia, August 18-26), ACM (2013) 389–399
10. Beyer, D., Keremoglu, M., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Formal Methods in Computer-Aided Design, 2010. FMCAD 2010. (2010)
11. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013, Rome, Italy, March 20-22). LNCS 7793, Springer-Verlag, Heidelberg (2013) 146–162
12. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. Proc. CAV, LNCS **1855** (2000) 154–169
13. Mandrykin, M., Mutilin, V., Khoroshilov, A.: Vvedenie v metod CEGAR – utochnenie abstraktsii po kontrprimeram [Introduction to CEGAR – Counter-Example Guided Abstraction Refinement]. Trudy ISP RAN [Proceedings of ISP RAS] **24** (2013) 219–292
14. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2012, Cary, NC, November 10-17), ACM (2012)
15. Mordan, V., Novikov, E.: Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules. Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering **8** (2014)
16. Zakharov, I., Mandrykin, M., Mutilin, V., Novikov, E., Petrenko, A., Khoroshilov, A.: Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software **41**(1) (2015) 49–64