Towards Automated Static Verification of GNU C Programs*

Evgeny Novikov and Ilja Zakharov

Institute for System Programming of the Russian Academy of Sciences
novikov@ispras.ru, ilja.zakharov@ispras.ru

Abstract. Static verification based on such methods as Bounded Model Checking and Counterexample-Guided Abstraction Refinement aims at non-interactive formal proving of programs correctness against safety property specifications. To leverage existing tools for verification of a program one should prepare verification tasks first. In addition to a program fragment of a moderate size, each verification task has to contain a rather accurate model of its environment. To achieve high-quality results this model should be incrementally refined in accordance with checked safety properties. For verification of specific software, like Windows or Linux drivers, a few frameworks provide a convenient user interface and perform in an automated way generation of verification tasks, execution of static verification tools and preliminary processing of results. This paper presents a method for automated static verification of any program developed in the GNU C programming language and addresses the ongoing development of the Klever framework.

Keywords: Static Verification, Software Verification, Formal Specification, Environment Model, GNU C.

1 Introduction

Static verification allows both finding of specified safety property violations and proving of formal program correctness under certain assumptions. There are many advanced static verification tools like SLAM [1], CBMC [2], CPAchecker [3] implementing Bounded Model Checking [4], Counterexample-Guided Abstraction Refinement [5] and other methods. However, their practical application is rather limited. Despite many merits of static verification users can not leverage the tools out of the box for industrial software. Existing static verification frameworks dramatically simplify the workflow, but their scope is bound to particular kinds of programs.

We propose a new method for automated static verification of programs developed in the GNU C programming language. Currently, the method has been partially implemented within the Klever static verification framework.

* The reported study was partially supported by RFBR, research project No. 16-31-60097.

2 Static Verification Background

In this section we consider an interface, features and requirements of static verification tools from the perspective of users.

Though first static verification tools appeared at the beginning of the century, their developers have been forming an active solid community. One of the most important steps in this direction was organization of a series of annual competitions on software verification, SV-COMP[1]. SV-COMP 2012 attracted about a dozen of static verification tool developer teams from leading research centers from all over the world [6]. The number of participants steadily grows and already 32 teams participated in SV-COMP 2017 [7]. SV-COMP competition rules de facto introduce a standard static verification tools interface accepted by all participants[2].

2.1 Target Programs

In this paper we consider primarily static verification of programs developed in the GNU C programming language. C programs verification is of great importance since there is much such software that operates in critical domains. For instance, various operating system kernels and libraries, programming language compilers and interpreters, database systems and web servers belong to this class.

Static verification tools analyze a model of a given program extracted automatically directly from its source code. To check software developed in various programming languages tools translate programs into internal representations using appropriate front-ends first. For instance, having a program LLVM internal representation, SMACK translates it into Boogie to run static verification tools like Corral then [8]. This potentially enables support for programming languages such as C and Java, since there are appropriate LLVM front-ends. The CPAchecker static verification tool has an internal representation suitable for checking C and Java programs [3]. For parsing C programs it uses the Eclipse CDT parser.

There is the SV-COMP benchmark suite to estimate and to compare capabilities of static verification tools comprehensively. According to the competition rules this suit contains so called verification tasks. A *verification task* contains a program and a safety properties specification. The safety properties specification represents requirements to check and we discuss this in the next subsection.

The program should be already prepared in advance so that static verification tools can take it as input without any additional processing and check it non-interactively. Programs developed in the GNU C programming language constitute the lion share of the competition benchmark suit. Thus, many participating in the SV-COMP competition static verification tools have a high-quality support of GNU C programs. Each program of the SV-COMP benchmark suite

[1] https://sv-comp.sosy-lab.org/
[2] https://sv-comp.sosy-lab.org/2017/rules.php

should be a single preprocessed C file. Just few static verification tools support verification tasks that consist of several source files.

2.2 Checking Requirements

In this paper we focus on static verification of programs against non-functional requirements. These requirements include generic rules of safe programming suitable for any program. Violations of these rules cover such errors as buffer overflows and null pointer dereferences. Also, we consider as non-functional requirements specific rules of correct usage of the particular program API which violations are also quite widespread [9]. Below we refer to both generic and specific rules as *correctness rules*.

Static verification tools can check safety and liveness properties. To run a tool the SV-COMP community first proposes to describe a safety properties specification as a temporal logic formula. Currently, the formula can represent only a hard-coded set of safety properties corresponding mostly to generic rules of safe programming:

- error function unreachability,
- valid memory deallocations, pointer dereferences and memory tracking (to search such errors as buffer over-reads and over-writes, null pointer dereferences, uses after free and memory leaks),
- absence of integer overflows,
- program termination.

For verifying other correctness rules a user can use the following two options. The first one is weaving an additional code into a program either manually or automatically to express requirements using one of the supported safety properties. For instance, correct API usage rules can be formulated as unreachability of an error function. Exploiting means of particular static verification tools is the second option. For instance, CPAchecker allows to specify requirements to a program using automata that guide analysis [10]. However, this approach prevents using other static verification tools and it is out of the scope of this paper. Also in addition to the SV-COMP safety property specifications some tools support checking of other requirements such as absence of data races [11].

Static verification tools allow checking safety property specifications one by one and most of them terminates after finding a first violation. That is why it is better to check substantially different correctness rules independently. Nevertheless, it makes sense to express closely related requirements using the same safety property to avoid analysis of similar results, say, false alarms due to the same reasons, and to decrease computational resources consumption.

2.3 Analysis Accuracy

Static verification tools construct a program model in a sound way that keeps all errors present in a source code under verification. Also, they check specified

safety properties at all possible paths including poorly tested ones. Due to this proving model correctness means that there is no erroneous paths in the program.

However, to deal with real programs developed in the GNU C programming language static verification tools usually make some assumptions. Many tools can treat undefined functions as functions without side effects returning any value of corresponding return types. According to our knowledge, no static verification tool has support of the inline assembler. Some tools, e.g. implementing Bounded Model Checking, unroll loops to a given number of iterations. These assumptions can result in both missing bugs and false alarms, but hopefully users can change parameters and provide models to bypass these issues.

2.4 Performance and Scalability

Static verification is an extremely complicated problem and even the best static verification tools have poor scalability at verification of large programs. Usually it is difficult to predict computational resources required for verification since an outcome depends on many factors such as a code complexity, a safety property being checked, verification algorithms, SMT and interpolation solvers. At the SV-COMP competition each verification task is executed under the following limits: 15 minutes of CPU time and 15 GB of RAM. Most successful tools can cope with programs of several dozens of KLOC in size within these limits but not always. Significant increasing of the code complexity almost always results in an enormous growth of required computational resources and inability to proceed with the same verification scope and precision.

Static verification tools often implement algorithms sequentially because of their nature, a few tools can use multi-core CPUs or distributed computing and there is no tools that employ GPUs [12]. Parallel solving of a verification task helps to yield verdicts faster but usually there is a considerable overhead to share complicated internal data structures. Thus, we do not consider this use case in the given paper. To substantially speed up solution of many independent verification tasks, static verification tools are executed in parallel at IaaS or PaaS clouds and clusters [12].

2.5 Environment Modeling and Checking Program Fragments

Libraries that used by a program, other programs, user inputs, etc. constitute an environment that can influence a program execution. To verify the program it is necessary to provide an environment model to a static verification tool which represents certain assumptions about the environment. In practice it should call program functions, initialize and modify a heap and exported variables as the environment does. Also, the environment model should contain models of undefined functions which the program calls.

Our experience shows that usually some results can be obtained even without the very accurate environment model. To get an acceptable false alarm rate and sufficient code coverage to avoid missing bugs it can be necessary to refine it.

To achieve really high-quality results it is crucial to provide the precise environment model taking into account specifics of the checked safety property and the program under verification.

To drastically reduce consumption of computational resources it is possible to verify program fragments of a moderate size separately. A program fragment can contain several source files of the program and libraries or just particular functions from them. However, it becomes even more important to provide the appropriate environment model at verification of the program fragment to avoid missing bugs and false alarms.

The SV-COMP community does not provide any tools or formats for facilitating specifying the environment model in a commonly accepted way. So, the most generic approach to provide the environment model is to weave it as an additional C code into the target program or the program fragment. Particular tools can provide additional means to describe an environment model as annotations, formulas or automata [10, 13]. But we do not consider such the use case in the given paper.

2.6 Tools Execution

Each verification task solution may require a considerable amount of CPU time, RAM and disk space. The BenchExec benchmarking tool comes to the aid as it enables fair computational resources distribution, isolates tool runs, performs preliminary results processing and simplifies integration of new static verification tools that follow the SV-COMP rules [14]. BenchExec measures and reports consumed CPU time, RAM and disk space. If tools exceed allowed limits, it terminates them. BenchExec is the convenient and reliable tool to solve verification tasks using a single machine.

BenchExec contains wrappers for all static verification tools that participate in the SV-COMP competition. Wrappers allow abstracting from the static verification tools interface when describing sets of verification tasks and when processing obtained results. However, a user still has to provide parameters to tune algorithms for each tool individually because default parameters defined at tool wrappers often do not suit for practical applications. For each successful static verification tool run a corresponding wrapper provides a verdict meaning a checked safety properties specification is satisfied or not. For each failed run the wrapper provides a short failure reason like "timeout" or "parsing failure" while details are kept within log files.

2.7 Formal Confirmation and Manual Analysis of Results

Static verification tools can provide proofs and counterexamples in a machine readable format on each successful run. There is a common format of correctness and violation witnesses[3]. The proposed witness validation technique establishes confirmation of such witnesses detecting spurious ones [15, 16]. The technique is

[3] https://github.com/sosy-lab/sv-witnesses

widely used in SV-COMP, so today all static verification tools participating in the competition provide their results similarly.

Witnesses express proofs and counterexamples using observer automata. A violation witness contains an observer automaton corresponding to a counterexample error path that leads from a program start to a found error. By design this automaton can miss some details of the error path and even some its parts. A witness validation tool considers the observer automaton in combination with a control-flow automaton extracted from the program to check feasibility of the error path.

Correctness witnesses have almost the same format but present results of proving, e.g. invariants for loops, for many paths of the program. However, correctness witnesses do not contain any complete proofs or guaranties for users. For instance, it is even impossible to understand which parts of the program were verified.

Although witnesses can be automatically validated, users should investigate them manually to comprehend proofs and reasons of bugs and false alarms. According to our knowledge, just CPAchecker represents witnesses in a more user-friendly way, but this does not help much for large programs because of this representation contains too many details. These details could be hidden using a domain knowledge, e.g. that some statements correspond to the environment model or some statements are not relevant for the checked safety properties specification.

In addition to witnesses some static verification tools can provide code coverage which lists analyzed lines and functions of the program. For its visualization one can use standard tools like LCOV[4].

BenchExec can provide to a user statistics on verdicts provided by static verification tools and on computational resources consumed by them in the form of tables and plots. Such results visualization finely presents performance and correspondence of obtained verdicts to ideal ones (ideal verdicts for SV-COMP verification tasks are known in advance). This suits pretty well for comparison of static verification tools at the competition. However, users need additional means convenient at practical use for evaluation of results obtained for their programs when ideal verdicts are unknown.

3 Automating Static Verification of GNU C Programs

We do not suggest solutions that allow to completely automate the static verification workflow from scratch for all programs developed in the GNU C programming language. Our primary goal is to suggest solid foundations that are generic enough to not restrict application to some specific software. However, the framework will support out of the box those program subclasses for which we or others will develop and implement appropriate algorithms. For other software it will be worth doing that to reduce required manual efforts, to increase results quality and to decrease demands for computational resources.

[4] http://ltp.sourceforge.net/coverage/lcov.php

Following subsections consider steps of the proposed method. For first two steps we implicitly assume that everything that is generated automatically can be incrementally refined manually if one will find this necessary.

3.1 Decomposition of Programs

Most of industrial programs are quite large, so it can be hard or impossible to statically verify them against any non-trivial correctness rule under reasonable computational resource limitations. To increase chances to get a meaningful outcome we suggest decomposing target programs and perhaps libraries invoked by them into program fragments which were already introduced in the previous section. The most challenging problem at programs decomposition is to determine particular files for program fragments. Program fragments may vary from a single C file to all files of a program or even several programs and libraries.

We suggest implementing the following options in the static verification framework:

- Provide to a user means to describe program fragments explicitly. This is the most time-consuming but flexible approach that can help to gain the best results. It is especially suitable at verifying some particular version and configuration of a medium-sized program.
- Develop a generic algorithm that will automatically split a program into weakly connected parts of a specified size. In contrast to the previous approach this one considerably reduces manual work at the programs decomposition step. But it might be necessary to spend much more time for analysis of worse results.
- Implement an algorithm for a particular project taking into account its structure. This approach requires an extra time for implementation for each new kind of programs but it contributes to both automatic programs decomposition and good enough results. This way suits for verification of large programs or many various configurations and versions of the same program.

For both manual and automatic approaches for program fragments generation it is useful to extract a build commands base that helps to understand how program source files are combined together to form the final object files and executables. For particular version and configuration of the program this base should include the following information on build commands in the strict order of their execution:

- For all build commands of interest:
  - corresponding build tool names,
  - absolute paths to directories where they are executed,
  - input and output file names.
- For compilation commands:
  - corresponding versions of source files referred both explicitly (C files) and implicitly (header files),
  - preprocessor options.

For extracting the build commands base we propose to intercept build commands during a program build. Indeed, GNU C programs often have complicated build processes. Thus, it can be necessary to describe semantics of additional build commands that should be intercepted.

The build commands base can be collected either outside the framework or from within it. The former is preferable when users want to perform builds as they usually do and to incorporate static verification with continuous integration systems. The latter is better when users need to verify some particular versions and configurations of their software.

3.2 Verification Tasks Generation

As it was stated in the previous section a verification task should contain besides a program fragment an additional code that corresponds to an environment model and, if necessary, that expresses checked requirements using one of the supported safety properties. Also, particular correctness rules can require setting specific parameters for a chosen static verification tool. We propose to generate these additional code and tool parameters on the basis of specifications developed manually using appropriate domain specific languages.

For each particular pair of a program fragment and a correctness rule a set of specifications can be unique, but some specifications can be the same for different pairs. To avoid repeats during development of specifications we suggest to use templates. Also, we propose to reuse a generated code if it is the same for different program fragments or correctness rules.

We already proposed a method for generating an environment model part that invokes program fragment interfaces for Linux kernel modules [17]. Corresponding specifications allow describing complicated interactions for event-driven programs in a quite compact way. Also, we have been starting developing a more generic approach on the base of this method. For developing the additional code, that expresses checked requirements using one of the supported safety properties, and models of interfaces invoked by the program fragment we suggest using an aspect-oriented extension for the C programming language [18]. Corresponding specifications and tools allow weaving program fragments, e.g. redirect function calls and macro substitutions from the original source code to model functions.

The final steps of the verification tasks generation is preprocessing, that is usually performed together with weaving, and merging of preprocessed files together. For the latter we suggest to use CIL that is a source-to-source transformation tool allowing merging files developed in the GNU C programming language [19]. Besides, CIL performs many optimizations simplifying following analysis.

After all each verification task is a single GNU C file prepared for an immediate run of a static verification tool and a safety properties specification which comply with the SV-COMP rules. In addition, a name, a version and parameters of the given tool and an amount of computational resources that can be used for solution are specified.

To incorporate a domain knowledge within verification tasks, e.g. to distinguish the additional code from the original one and to emphasize statements that are the most relevant to checked requirements, we suggest to use special comments. These comments can be provided directly within specifications. Besides, they can be generated automatically.

3.3 Verification Tasks Solution and Results Processing

Verification tasks generation can take a considerable amount of time. Hence, we suggest to start solution of verification tasks as soon as they appear if there are enough computational resources. For solving a few verification tasks we propose to use a single powerful enough machine. To considerably reduce a total time for verification of many program fragments or/and correctness rules it is necessary to solve corresponding verification tasks at an IaaS cloud or at a cluster.

Monitoring of available computational resources and their fair distribution between verification tasks are responsibilities of a scheduler. The scheduler should respect verification task priorities specified by users. Also, the scheduler should support canceling solution of verification tasks since sometimes users can decide that they do not need to continue verification anymore.

Some verification tasks can require considerably less computational resources than requested by a user. To avoid useless reservation we suggest performing speculative scheduling trying to run a static verification tool with lesser limitations first. It is worth accumulating statistics for verification tasks solution to base scheduling on that ground. For instance, some correctness rules can be much easier for checking on average than other ones.

In case of using a cloud or a cluster the scheduler should allow connecting and disconnecting worker nodes. If a worker node goes down, we suppose to automatically reschedule terminated verification tasks solution.

For isolating static verification tool runs and for measuring and limiting computational resources consumed by them at a single machine we propose to use already mentioned BenchExec [14]. After BenchExec finishes, the framework should process its output and results from a static verification tool. We suggest retrieving a verdict, a consumed computational resources report, a witness, log files and other information like code coverage and statistics if so.

As far as witnesses can omit some details, we suggest adding them by considering witnesses together with the program from the corresponding verification task. Besides, we propose to enrich witnesses with domain knowledge annotations on the base of special comments generated at the previous step

Regarding code coverage users can be interested in total code coverage for all program fragments rather than code coverage for individual verification tasks. Thus, we suggest uniting it for various correctness rules.

3.4 User Interface

Below we present different use cases of the static verification framework and discuss relevant user interfaces.

Verification Processes Setup. In this use case we assume that the framework already supports everything required for a program under verification, in particular appropriate correctness rule specifications are available.

To proceed to verification users should choose correctness rules to be checked and provide program fragments either by describing them manually or by choosing and configuring an appropriate algorithm (a target program should be provided in form of its source code or build commands base). In addition, to get better results for particular programs we suggest to incrementally tune various parameters for programs decomposition, verification tasks generation, verification tasks solution and results processing.

For providing data and parameters and for starting subsequent automatic static verification we propose to use a multiuser graphical interface shareable via a network. Project specific interfaces, which assumes various forms and helpers, are most likely the most convenient way for users, but usually it is hard to develop them. Therefore, we suggest to support a file-based configuration that is flexible enough to cope with various programs. In addition, users should be able to start up verification using command-line tools providing some data like a build commands base. This use case is quite natural when one incorporates static verification with continuous integration systems.

Expert Results Analysis. To simplify analysis of results by experts we suggest extending the interface for verification processes setup with the following:

- Provide information on running and completed verification processes. For each verification process experts should be able to analyze data and parameters with which it was set up. For running verification processes the interface should present their progress: the number of already solved and the total number of verification tasks, elapsed time and approximate left time.
- Visualize witnesses, code coverage and failure descriptions. The primary goal of this visualization is to hide from experts as much irrelevant details as possible according to the domain knowledge.
- Show various statistics over results that can help to understand a picture in general. For instance, it can be very useful to see how many warnings were yielded for a particular verification process, what warnings correspond to bugs and to false alarms, what are the most significant reasons of false alarms and so on.
- Allow to evaluate results by associating them with marks that should be applied automatically for similar results such as witnesses and failure descriptions. Experts should be able to supply each mark with a detailed description and tags. To further simplify analysis we suppose to keep all history of marks changes.
- Support views allowing arranging data in a more convenient way and to filter out irrelevant results. For instance, experts may want to see just violations of a specific correctness rule or marks modified after some date.

It is worth noting that since static verification can take very considerable time it does have sense to represent results to experts as soon as they appear. In this case they are able to proceed to analysis of results faster, in particular it is possible to understand that something was done wrong without waiting for all results.

Developing Correctness Rule Specifications and Extending the Framework. If verification process setup itself does not help to improve results quality, e.g. to increase code coverage or to decrease a false alarm rate, we suggest to incrementally improve correctness rule specifications. In case when the static verification framework does not cope well with specific programs out of the box, one can develop and implement more appropriate algorithms to be incorporated into the framework.

Users can perform both these activities using their favorite editors or IDEs. Also, we encourage to supply them with special domain specific language editors and a SDK for developing framework extensions. We propose the framework GUI to support simple extensions like hot plugging of new static verification tools.

To understand consequences of improvements we suggest users should be able to compare results and associated expert marks for different verification processes.

4 Implementation

We partially implemented the proposed method for automated static verification of programs written in the GNU C programming language within the Klever framework[5]. Klever is an open source project. The primary programming language is Python 3.4. Users can install Klever on various Linux distributions. Also, we implemented scripts for automatic deployment within an OpenStack cloud.

We use the Django framework for developing *Klever Bridge* that provides the web GUI for verification processes setup and expert results analysis. As a database system *Klever Bridge* supports PostgreSQL and MariaDB. For deploying the GUI users can use either Apache2 with mod_wsgi or NGINX with Gunicorn. *Klever Bridge* already supports multiple users with different roles, fair results representation and automated results assessment. For the latter one can use one of the several algorithms for comparing violation witnesses and regular expressions for matching failure descriptions. At the moment *Klever Bridge* does not support visualization of correctness witnesses.

Users should specify for each cloud or cluster worker node how many CPU cores, RAM and disk space the framework can use for solving verification tasks. Enough computational resources should be reserved for generation of program fragments and verification tasks, results processing as well as for an operating system and other running services and applications.

There are three schedulers currently implemented:

[5] https://forge.ispras.ru/projects/klever

1. *Klever Native Scheduler* provides means to solve verification tasks using a single machine. For monitoring available computational resources and running services it uses Consul.
2. *Klever Docker Scheduler* can solve verification tasks within a cluster or a cloud by leveraging an infrastructure for Docker containers.
3. *Klever VerifierCloud Scheduler* submits verification tasks to VerifierCloud[6].

At the moment verification tasks can be solved with help of CPAchecker and Ultimate Automizer [20]. To integrate new static verification tools within Klever users need to do the following:

1. Describe specific options suitable for checking corresponding correctness rules.
2. Provide static verification tool binaries in case of using *Klever Native Scheduler*.
3. For *Klever Container Scheduler* install tools within Docker images and push these images to a Docker registry.

Currently, fully automatic programs decomposition and generation of verification tasks is available only for Linux kernel loadable modules as a proof of concept. The framework extracts the build command base itself allowing users to guide the build process via parameters. Users can choose one of the several algorithms for program fragments generation. The main one is to verify each Linux kernel loadable module separately. Another one allows specifying files and modules to unite manually. It is also possible to generate program fragments for groups of modules fully automatically on the base of dependencies between modules using a greedy algorithm.

The framework generates verification tasks in parallel to speed up the entire verification process when many computational resources are available. As a source code querier and a weaver we use CIF [18]. It is a source-to-source weaver that is based on GCC, and, thus, it can handle GNU C programs. CIF allows to perform a variety of structural source code queries and support weaving of macro substitutions and function calls.

The environment model is described as a parallel composition [17]. It is translated into a C code using an additional information extracted by querying a source code of a target program. Utilization of different translators allows generating either a parallel or sequentialized environment model. It also allows making heuristic simplifications of the model, e.g. to reduce interleaving. For Linux kernel loadable modules we have implemented environment model specifications to support interrupts, timers and interfaces of kernel subsystems including USB, PCI, SCSI, SERIAL, NET, file systems, etc.

We allow users to check a variety of correctness rules ranging from generic memory safety to correct usage of the most popular Linux kernel API. Also, one can check new requirements by developing additional specifications.

[6] https://vcloud.sosy-lab.org/cpachecker/webclient/help/

5 Related Work

According to our knowledge, no static verification framework automates preparation of an arbitrary GNU C program before static verification, runs static verification tools, processes results and provides means for their further analysis and improvement. There is a few projects that focus on automated static verification of specific software.

SDV is the best-known application of static verification in practice [21]. It aims at checking correct usage of the kernel API in Windows drivers using SLAM, YOGI and Q [1, 22]. There are also LDV Tools [23], DDVerify[24], Avinux [25] frameworks intended for static verification of Linux drivers. As a result, hundreds of bugs have been found and acknowledged by driver developers already.

CBMC [2] has been applied for verification of TinyOS [26] and embedded software [27]. Authors deliver successful case studies as a proof of concept.

DC2 is a framework that aims at static verification of industrial software [13]. To bound a verification scope it generates contracts relevant for safety properties like memory leaks and array-bound overflows. If necessary, users can improve these contracts manually. Then DC2 runs the Varvel model checker. However, it is the in-house NEC research project, so it is not possible to estimate its applicability to software developed in the GNU C programming language in more details.

There is an IDE for development of embedded software *mbeddr* that allows to automatically run CBMC to check programs under development against a predefined set of safety properties [28]. The IDE also provides developers with nicely arranged results. However, *mbeddr* is not intended for automated static verification of programs developed outside of it.

6 Conclusion

We presented the method that addresses problems of automated application of static verification tools for checprograms developed in the GNU C programming language. This method has been partially implemented within the Klever framework that already demonstrated its applicability to large industrial software projects like Linux kernel loadable modules. To complete the research, we are going to provide comprehensive evaluation verifying various programs.

We based Klever on solutions accepted by the SV-COMP community. Moreover, we keep in touch with it to cooperate and to solve the most vital problems together discussing the interface, providing a feedback and contributing generated verification tasks to the competition benchmark suit.

References

1. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, Austin, TX, FMCAD Inc (2010) 35–42

2. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In Jensen, K., Podelski, A., eds.: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer (2004) 168–176
3. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification, Berlin, Heidelberg, Springer (2011) 184–190
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 117–148
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM **50**(5) (2003) 752–794
6. Beyer, D.: Competition on software verification. In Flanagan, C., König, B., eds.: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer (2012) 504–524
7. Beyer, D.: Software verification with validation of results. In: Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer (2017) 331–349
8. Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: SMACK+Corral: A modular verifier (competition contribution). In Baier, C., Tinelli, C., eds.: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer (2015) 450–453
9. Mutilin, V.S., Novikov, E.M., Khoroshilov, A.V.: Analysis of typical faults in Linux operating system drivers. Proceedings of ISP RAS **22** (2012) 349–374
10. Apel, S., Beyer, D., Mordan, V., Mutilin, V., Stahlbauer, A.: On-the-fly decomposition of specifications in software model checking. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, ACM (2016) 349–361
11. Andrianov, P.S., Mutilin, V.S., Khoroshilov, A.V.: Predicate abstraction based configurable method for data race detection in Linux kernel. In: Proceedings of the 4th International Conference on Tools and Methods of Program Analysis, Springer (2017)
12. Zakharov, I.S.: A survey of high-performance computing for software verification. In: Proceedings of the 4th International Conference on Tools and Methods of Program Analysis, Springer (2017)
13. Ivančić, F., Balakrishnan, G., Gupta, A., Sankaranarayanan, S., Maeda, N., Imoto, T., Pothengil, R., Hussain, M.: Scalable and scope-bounded software verification in Varvel. Automated Software Engineering **22**(4) (December 2015) 517–559
14. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In Fischer, B., Geldenhuys, J., eds.: Proceedings of the 22nd International Symposium on Model Checking Software, Cham, Springer (2015) 160–178
15. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, ACM (2015) 721–733
16. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, ACM (2016) 326–337

17. Khoroshilov, A., Mutilin, V., Novikov, E., Zakharov, I.: Modeling environment for static verification of Linux kernel modules. In Voronkov, A., Virbitskaite, I., eds.: Proceedings of the 9th International Conference on Perspectives of System Informatics, Berlin, Heidelberg, Springer (2015) 400–414
18. Novikov, E.M.: An approach to implementation of aspect-oriented programming for C. Programming and Computer Software **39**(4) (2013) 194–206
19. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In Horspool, R.N., ed.: Proceedings of the 11th International Conference on Compiler Construction, Berlin, Heidelberg, Springer (2002) 213–228
20. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: Ultimate Automizer with array interpolation. In Baier, C., Tinelli, C., eds.: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Berlin, Heidelberg, Springer (2015) 455–457
21. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Communications of the ACM **54**(7) (2011) 68–76
22. Lal, A., Qadeer, S.: Powering the Static Driver Verifier using Corral. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, ACM (2014) 202–212
23. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E.M., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software **41**(1) (2015) 49–64
24. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: Proceedings of the 22nd International Conference on Automated Software Engineering, New York, NY, USA, ACM (2007) 501–504
25. Post, H., Küchlin, W.: Integrated static analysis for Linux device driver verification. In: Proceedings of the 6th International Conference on Integrated Formal Methods, Berlin, Heidelberg, Springer (2007) 518–537
26. Bucur, D., Kwiatkowska, M.Z.: Software verification for TinyOS. In: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, New York, NY, USA, ACM (2010) 400–401
27. Schlich, B., Kowalewski, S.: Model checking C source code for embedded systems. International Journal on Software Tools for Technology Transfer **11**(3) (2009) 187–202
28. Cârlan, C., Ratiu, D., Schätz, B.: On using results of code-level bounded model checking in assurance cases. In Skavhaug, A., Guiochet, J., Schoitsch, E., Bitsch, F., eds.: Proceedings of the 2016 International Conference on Computer Safety, Reliability, and Security, Cham, Springer (2016) 30–42