DOI: 10.15514/ISPRAS-2021-33(6)-1

Capabilities and Restrictions of Software Model Checkers

E.M. Novikov, ORCID: 0000-0003-3586-3140 <novikov@ispras.ru>
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Software model checkers enable automatic detection of violations of specified requirements in programs as well as formal proof of correctness under certain assumptions. These tools actively evolve two last decades. They were already successfully applied to a bunch of industrial projects, first of all to kernels and drivers of various operating systems. This paper considers an interface of software model checkers, their unique capabilities as well as restrictions that prevent their large-scale usage on practice.

Keywords: software model checking; formal verification; requirements specification; violation witness.

For citation: Novikov E.M. Capabilities and Restrictions of Software Model Checkers. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 6, 2021, pp. 7-14. DOI: 10.15514/ISPRAS–2021–33(6)–1

Возможности и ограничения инструментов верификации моделей программ

Е.М. Новиков, ORCID: 0000-0003-3586-3140 <novikov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Инструменты верификации моделей программ позволяют автоматически искать нарушения специфицированных требований в программах, а также доказывать их корректность формально при выполнении определенных условий. Данные инструменты развивались достаточно активно два последних десятилетия. За это время они были успешно использованы в ходе верификации нескольких промышленных проектов, в первую очередь ядра и драйверов различных операционных систем. Данная статья рассматривает интерфейс инструментов верификации моделей программ, их уникальные возможности, а также ограничения, которые затрудняют их широкомасштабное практическое применение.

Ключевые слова: верификация моделей программ; формальная верификация, спецификация требований; свидетельство нарушения

Для цитирования: Новиков Е.М., Возможности и ограничения инструментов верификации моделей программ. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр. 7-14. DOI: 10.15514/ISPRAS–2021–33(6)–1

1. Introduction

Software model checking helps to find violations of specified requirements in programs non-interactively and prove program correctness formally under certain assumptions. These capabilities are highly demanded by the industry since conventional quality assurance approaches either fail to reveal all faults in programs under verification [1] or their usage needs enormous efforts [2, 3].

Developers of software model checkers (they will be also called *verification tools* below) have been forming an active community since the appearance of first such tools at the beginning of the century. One of the most important steps in this direction was the organization of a series of annual competitions on software verification (SV-COMP). The first competition in 2012 attracted about a dozen of developer teams from leading universities and research centers from all over the world [4]. Since then, the number of participants has been growing steadily and already 27 teams participated in SV-COMP 2021 [5].

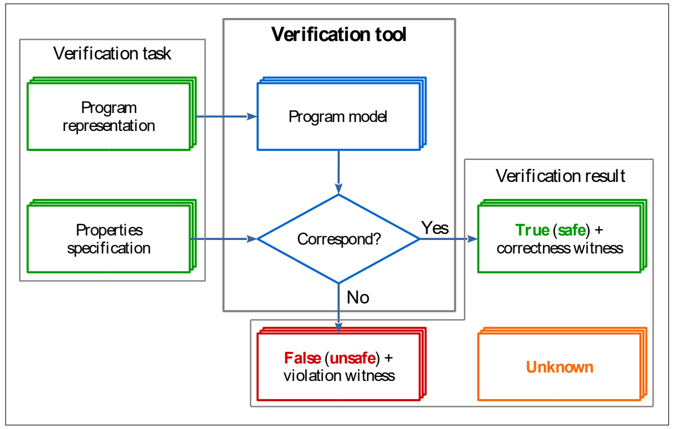

Fig. 1. Verification tool interface and basic workflow

Fig. 1 illustrates an interface and a basic workflow of a verification tool. The interface is specified in detail in SV-COMP rules [6]. Though, it changes from year to year but highly likely most major decisions have been done already. The verification tool gets as an input a so-called *verification task* that consists of a program representation, e.g. files with program sources or LLVM bitcode, and a properties specification. The former is based on the source code of the target program and the latter allows to specify requirements to check. The verification tool builds a program model on the base of the program representation and checks it against the properties specification. This is performed fully automatically.

As a result the verification tool provides a *verdict* that answers the following question: «Does the program satisfy the specification?» In case of successful verification, it also outputs a *witness* in addition to the verdict. Witnesses are machine-readable files containing parts of formal evidences that can be validated automatically or studied manually to confirm or to reject verification results. If the verification tool cannot provide a definite answer, say, because it depletes allotted computational resources such as CPU time or memory, then it gets terminated and the verdict is set to *unknown*.

Following sections consider particular aspects of the verification tool interface, capabilities and restrictions of verification tools as well as extra features required for application of them to industrial programs.

2. Supported Programs

Verification tools support verification of software developed in various programming languages. This paper considers verification of C programs exclusively since they are quite widespread among industrial programs requiring a very high level of quality.

According to SV-COMP rules a verification task should contain a single C source file. This file should be prepared in advance so that verification tools can take it as input without any additional processing. Since industrial programs usually contain many source files, users should preprocess and merge them beforehand. Most programs interact with their environment including users, libraries, other programs, hardware and so on. This interaction should be incorporated into the C source file of the verification task since verification tools operate non-interactively, they do not communicate to program environments and they dislike undefined behavior. SV-COMP rules do not consider how this can be achieved. This topic will be considered further in Section 6.

Programs developed in GNU C constitute the lion share of the SV-COMP benchmark suite with the combined size of approximately 100 MLOC of preprocessed code. Thus, many verification tools that participate in SV-COMP have a high-quality support for GNU C programs, though some specific extensions, e.g. attributes, can be unsupported. Support for other compiler extensions depends on used front-ends primarily.

On average verification tasks are 3 KLOC in size. That's why users can hardly expect that they will be able to use verification tools out of the box for industrial programs that contain hundreds and thousands of KLOC. The SV-COMP community does not provide users with ideas and auxiliary tools to tackle the given issue. In following sections performance and scalability of verification tools will be considered in more details.

Many industrial C programs use various parallel programming means. This paper treats just multithreading. Accurate verification of multithreaded programs is a much harder task in comparison with verification of sequential programs. Later a special attention will be paid to the given issue.

3. Supported Requirements

This paper focuses on verification of programs against non-functional requirements which violations can result in critical failures like denial of service, privilege escalation and data breaches[1]. For instance, it is vital to detect such common weaknesses of C programs as buffer overflows and null pointer dereferences. Other examples of non-functional requirements are rules of correct usage of an API, which are also often violated and which violations can be rather harmful [7].

Table 1. Formulas describing properties supported by verification tools

Property	Formula
Unreachability	*CHECK(init(main()), LTL(G ! call(reach_error())))*

Memory Safety	*CHECK(init(main()), LTL(G valid-free))*
	CHECK(init(main()), LTL(G valid-deref))
	CHECK(init(main()), LTL(G valid-memtrack))
	CHECK(init(main()), LTL(G valid-memcleanup))

Overflow	*CHECK(init(main()), LTL(G ! overflow))*
Termination	*CHECK(init(main()), LTL(F end))*

Verification tools can check programs against safety and liveness properties. SV-COMP defines a common format for specifications of such properties in the form of linear temporal logic (LTL) formulas. Table 1 presents currently supported properties and corresponding formulas. Here *init(main())* represents a program entry point assuming calling function *main()* without parameters. LTL operator *G f* means that formula *f* holds in every state of the program, so, for example, *G ! overflow* means that integer overflow should never happen, and *G ! call(reach_error ())* means that the error function should not be ever called (otherwise, there may be faults in checked programs). If unclear, semantics of other properties and formulas can be clarified using SV-COMP rules.

[1] For thorough checking of program functionality other methods, such as deductive verification, suit better. Similarly, for detecting non-critical non-functional requirements, e.g. coding style violations, there are other good methods and tools.

The SV-COMP community does not suggest any widely accepted means for checking those requirements that do not correspond explicitly to one of the supported properties. Users can either leverage specific capabilities provided by some verification tools, e.g. for finding data races [8], or weave an additional source code into a program either manually or automatically to express requirements using one of the supported properties. For instance, rules of correct usage of a particular API can be formulated as unreachability of the error function like in Fig. 2 and Fig. 3.

```
/* Device driver can register just
   one device at a time. */
int register_device(void) {
    ...;
}

/* Device driver can unregister device
   just after it registers it. */
void unregister_device(void) {
    ...;
}
```

Fig. 2. Original program

```
bool is_device_registered = false;

int register_device(void) {
    if (is_device_registered)
        reach_error();
    is_devi)ice_registered = true;
    ...;
}

void unregister_device(void) {
    if (!is_device_registered)
        reach_error();
    is_device_registered = false;
    ...;
}
```

Fig. 3. Woven in program

If one expresses weakly related requirements using the same property, it is possible to check them simultaneously, but this is not recommended due to the following issues. The first reason for this is that verification tools may build and check substantially different models. It is not an easy task how to distribute available computational resources between these models when they are complex enough. The second reason is that most verification tools stop after they find a first violation of a checked property. So, detecting a first fault or a false alarm can prevent finding other faults. Overall, it may be better to check different requirements independently.

Verification tools can hardly check large and complicated parallel programs. Hopefully, for checking most of requirements it is not necessary to consider all possible interleavings of threads[2]. There are different approaches how to serialize parallel programs. Section 6 presents some related ideas.

4. Verification Accuracy

Verification tools tend to construct program models in a sound way that keeps all errors existing in the source code under verification. However, as a rule they make some assumptions either implicitly or explicitly according to specified configuration options to significantly raise their efficiency. For

[2] Here it is assumed that the target program does not have any concurrency issues.

example, many verification tools can treat undefined functions, e.g. library functions, as functions without side effects returning any value corresponding to their return types. Often this does not affect verification results[3], but sometimes this is not the case, e.g. when undefined functions allocate and initialize memory referred later. Most verification tools do not support the inline assembler. Verification tools implementing bounded model checking unroll loops just to a given number of iterations.

All these assumptions can result in both missing faults and false alarms. Fortunately, verification tools can report possible verification inaccuracies while users can change corresponding configuration options and provide models to bypass these issues at least to some extent.

5. Performance and Scalability

Comprehensive verification is an extremely complicated problem. Usually, it is difficult to predict computational resources necessary for it since an outcome depends on many factors such as code complexity, requirements being checked, verification algorithms, solvers and so on.

SV-COMP rules specify the following limits for each verification task: 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 increase of complexity of source code parts relevant to checked requirements almost always results in an enormous growth of required computational resources and inability to proceed with the same verification scope and precision. This is demonstrated by three quantile plots in Fig. 4. These plots show how many verification tasks can be solved using an appropriate amount of CPU time. One can see that verification tools operate differently but none of them can solve all verification tasks within the specified time limit.

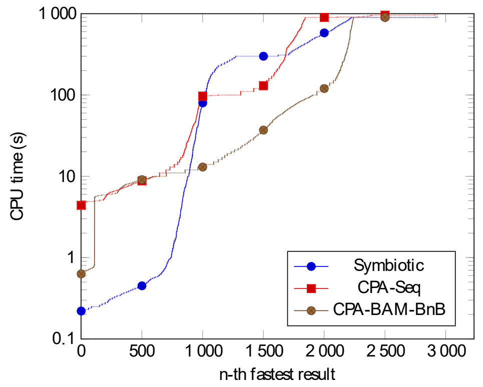

Fig. 4. Statistics for winners of SV-COMP'20 in the «Software Systems» category

Verification tools often implement algorithms sequentially because there is a considerable overhead to share complicated internal data structures. A few tools can use multi-core CPUs or distributed computing and there are several tools that employ GPUs. To substantially speed up solution of many independent verification tasks, verification tools are executed in parallel at IaaS or PaaS clouds and clusters. You can find more information about this in the appropriate survey [9].

6. Environment Modeling and Checking Program Fragments

Libraries, user inputs, other programs, etc. constitute an environment that can influence a program execution. To verify the program, it is necessary to provide a model which represents certain assumptions about the environment:

[3] Indeed, programs should carefully inspect return values of called functions if so since often they can represent error codes.

- The environment model should invoke a program API in a way the environment can do.
- It should contain models of undefined functions which the program calls during execution and which can influence verification results for checked requirements.

Bug finding is possible even without very accurate environment models. Still, more accurate environment models help to improve code coverage and reduce a false alarm rate. To achieve high-quality verification results it is crucial to provide the precise environment model taking into account specifics of checked requirements and programs under verification.

At environment modeling it is important to distinguish parallel and sequential cases. It is pretty natural to have a parallel environment model that can accurately reflect all possible interactions with the real environment. Unfortunately, as it was already mentioned, it may be too hard for verification. Thus, one has to provide sequential environment models and verify target programs with them. For instance, for libraries defining a set of functions it is possible to invoke them one by one[4]. For event-driven programs one can invoke callbacks just after their registration is completed or upon appropriate events are triggered by target programs [10].

To drastically reduce consumption of computational resources and increase chances to obtain verification results in a reasonable time, one can 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.

It becomes even more important to provide the appropriate environment model to avoid missing faults and false alarms at verification of such program fragments. Decomposing a program into individual C source files or even particular functions, which is an obvious way to simplify verification tasks, can require enormous efforts for modeling the environment. From common sense and from practical experience one needs to treat logically interconnected program components as program fragments like, say, loadable kernel modules or plugins (Table 2). Often this is a «golden mean» that enables obtaining useful verification results with moderate efforts for modeling the environment.

Table 2. Approximate number of components in open source projects

Project	Number of components
Linux kernel	5000
BusyBox	300
GTK library	200
Apache HTTP Server	150
VLC media player	80

The SV-COMP community does not propose any commonly accepted means for decomposing large programs into fragments and for specifying the environment model.

7. Formal Confirmation and Manual Analysis of Verification Results

After each successful run, verification tools provide proofs (*correctness witnesses*) and counterexamples (*violation witnesses*) in a machine-readable format [11]. The proposed witness validation technique establishes confirmation of such witnesses detecting spurious ones [12, 13]. The technique is widely used in SV-COMP, so today all verification tools participating in the competition can provide witnesses. This paper considers just violation witnesses since correctness witnesses serve primarily for validation and cannot help expert to comprehend proofs.

A violation witness describes a subset of paths from an entry point to a found error. By design, it can miss some details and even some parts of these paths. A witness validation tool considers the

[4] The same functions can be invoked multiple times, but still sequentially.

violation witness in combination with a control-flow automaton extracted from the program representation to recheck the solution of the verification task.

Although violation witnesses can be automatically validated, users still need to investigate them manually to understand reasons of faults and false alarms (false alarms can be caused by either imprecision of verification tools or environment models). There are some tools for visualization of witnesses in a more user-friendly way, but they do not help much for large programs because visualizations can contain too many details irrelevant for checking particular requirements. Moreover, experts need means for assessing verification results obtained for different versions and configurations of target software.

Some verification tools, e.g. CPAchecker [14], can provide code coverage reports in addition to witnesses [15]. These reports are in the GCC test coverage format (GCOV). For its visualization one can use standard tools like LCOV [16].

Code coverage reports are an important artifact to establish verification in practice. They reflect parts of the program such as lines of code, branches and functions that are actually verified. This information is essential for estimating an environment model quality since neither violation nor correctness witnesses do not provide data on actually considered program paths. Code coverage reports help to understand which program entry points should be invoked additionally by environment models. SV-COMP does not focus on this useful artifact.

8. Conclusion

Verification tools participating in SV-COMP demonstrate excellent results for verification tasks included into the benchmark suite. Some verification tools miss a few faults and report not so many false alarms. This works for moderate-sized programs that are prepared in advance and checked against the predefined list of properties.

Industrial C programs can contain much more source code than typical verification tasks. Also, during work they can extensively interact with their environments. This hinders or even makes impossible application of verification tools for them. Moreover, users can need to check specific requirements in addition to supported properties. At last, existing tools do not provide users with a comprehensive enough suite of means for analysis of verification results.

To reduce efforts that are necessary for application of verification tools for large industrial C programs one develops tools and infrastructures around them, e.g. SDV [17], Klever [18, 19] and VerifierCloud [20]. They provide very different capabilities and look very diversely. Often they are intended just for a particular type of programs and bound with the only verification tool. Thus, large-scale application of software model checkers still requires more research and development to cope with all their restrictions and get all expected benefits.

References

[1]. P.E. Black, L. Badger et al. Dramatically reducing software vulnerabilities: report to the White House Office of Science and Technology Policy. Technical Report NISTIR 8151, National Institute of Standards and Technology, Gaithersburg, MD, 2016.

[2]. G. Klein, J. Andronick et al. Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems, volume 32, issue 1, 2014, pp. 1-70.

[3]. D. Efremov, M. Mandrykin, A. Khoroshilov. Deductive verification of unmodified Linux kernel library functions. Lecture Notes in Computer Science, vol. 11245, 2018, pp. 216-234.

[4]. D. Beyer. Competition on software verification. Lecture Notes in Computer Science, vol. 7214, 2012, pp. 504-524.

[5]. D. Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). Lecture Notes in Computer Science, vol. 12652, 2021, pp. 401–422.

[6]. Definitions and Rules of 10th International Competition on Software Verification. URL: https://sv-comp.sosy-lab.org/2021/rules.php, accessed 12.12.2021.

[7]. V. Mutilin, E. Novikov, A. Khoroshilov. Analysis of typical faults in Linux operating system drivers. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 349-374 (in Russian). https://doi.org/10.15514/ISPRAS-2012-22-19.

[8]. P. Andrianov. Analysis of correct synchronization of operating system components. Programming and Computer Software, vol. 46, issue 8, 2020, pp. 712-730. https://doi.org/10.1134/S0361768820080022.

[9]. I. Zakharov. A survey of high-performance computing for software verification. Communications in Computer and Information Science, vol. 779, 2017, pp. 196-208.

[10]. I. Zakharov, E. Novikov. Compositional environment modelling for verification of GNU C programs. In: Proc. of the Ivannikov Ispras Open Conference, 2018, pp. 39–44.

[11]. Exchange Format for Violation Witnesses and Correctness Witnesses. URL: https://github.com/sosy-lab/sv-witnesses, accessed 12.12.2021.

[12]. D. Beyer, M. Dangl et al. Correctness witnesses: exchanging verification results between verifiers. In Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 326-337.

[13]. D. Beyer, M. Dangl et al. Witness validation and stepwise testification across software verifiers. In Proc. of the 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 721-733.

[14]. D. Beyer, M.E. Keremoglu. CPAchecker: a tool for configurable software verification. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 184–190.

[15]. R. Castaño, V. Braberman et al. Model checker execution reports. In Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 200-205.

[16]. LCOV – the LTP GCOV extension. URL: https://github.com/linux-test-project/lcov, accessed 12.12.2021.

[17]. T. Ball, V. Levin, S.K. Rajamani. A decade of software model checking with SLAM. Communications of the ACM, vol. 54, issue 7, 2011, pp. 68-76.

[18]. Klever: a software verification framework. URL: https://forge.ispras.ru/projects/klever, accessed 12.12.2021.

[19]. E. Novikov and I. Zakharov. Towards automated static verification of GNU C programs. Lecture Notes in Computer Science, vol. 10742, 2018, pp. 402-416.

[20]. VerifierCloud Web Interface. URL: https://vcloud.sosy-lab.org, accessed 12.12.2021.

Информация об авторах / Information about authors

Евгений Михайлович НОВИКОВ – кандидат физико-математических наук, ведущий научный сотрудник отдела Технологий программирования ИСП РАН. Его научные интересы включают верификацию моделей программ, спецификацию требований и операционные системы.

Evgeny Mikhailovich NOVIKOV – PhD, Senior Researcher at the Software Engineering Department of ISP RAS. His research interests include software model checking, requirements specification and operating systems.