Quantification of Verification Progress

[Extended Abstract]

Stephan Arlt
Université du Luxembourg

John Murray
SRI International

Philipp Rümmer
Uppsala University

Martin Schäf
SRI International

ABSTRACT

A key disadvantage of software verification over other quality assurance techniques, such as testing, is its unpredictable cost. A lot of people-hours have to be invested before correctness can be proved, and, in contrast to testing, there is no quantifiable evidence that incremental verification effort results in incremental quality improvements. On the other hand, the process of verifying code can be seen as a sort of audit or code-walk, so there is an intuition that the process itself improves quality, even before a proof can be computed. In this paper we discuss our first attempts to quantify the incremental quality improvements that are achieved during verification using a metric called verification coverage.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Software verification, coverage metrics

1. INTRODUCTION

So far, we see a very limited adoption of formal verification into the industrial software development process. This is largely due to the unpredictable return on investment that is inherent to verification. One has to invest a considerable amount of people-hours to prepare software for verification (e.g., by specifying, or annotating code) before attempting a proof of correctness, and one has no guarantee that this proof can be computed by existing tools given the undecidability of the problem. Until the proof is obtained there is no notion for the quality improvement gained through verification. That is, unlike, for example, in testing, there is currently no evidence that incremental verification effort results in incremental quality improvements.

Intuitively, we would argue that any verification effort improves software quality, regardless of if we reach an actual proof or not, and that more verification effort leads to better code quality. To a large extent, verification is like a code-walk or an audit. We walk through a piece of code line by line and try to understand what it does, and, in the case of verification, even try to formalize this behavior in a machine readable form. Hence, if audits and code-walks improve code quality, so should formal verification. What is missing is solid empirical evidence to support this.

Having some solid evidence that the attempt of verifying software gives quantifiable incremental quality improvements even if no proof is reached could pave the way for a broader industrial adoption of state of the art tools.

Recently, we have been looking into possible metrics to support the claim that incremental verification effort leads to incremental quality improvements. Such a metric has two essential requirements:

1. The metric has to reflect the progress of verification. That is, code that is sufficiently specified to be analyzed by state of the art verification tools has to be more highly ranked than code that is has insufficient or overly strong specification (e.g., preconditions that exclude part of the code).
2. The metric must be able to reflect code quality even if no specification is available. That is, given two versions of a programs source code, the metric must be able to detect if one version is more error prone than the other.

In a first attempt we started using the percentage of (implicit) assertions in a program that can be verified as a metric of **verification coverage** [1]. The term verification coverage is loosely based on the idea of test coverage, indicating that it represents the percentage of statements which we have explored exhaustively.

Using verification coverage as a measure of progress for verification is based on our experience with how code-walks are done: a piece of code is brought up on the screen and inspected line by line. For each line, it is discussed if this line might fail and under which precondition. From a verification point of view, this can be seen as trying to verify the implicit run-time assertions associated with that line and stating the necessary preconditions. Functional correctness of code that is often discussed in audits can be encoded using assertion statements.

Obviously, the problem of computing verification coverage is just as undecidable as the problem of proving correctness of the entire program. So the question that we need to discuss is if a reasonable approximation of verification coverage can be computed efficiently. Here, reasonable means that we need a suitable under-approximation of the actual verification coverage. An over-approximation would be dangerous, just like unsound verification, as it vouches for possibly incorrect code.

In the following, we discuss two experimental setups where we tried to compute verification coverage. We discuss how our approaches scale and where we had to sacrifice soundness.

2. DEDUCTIVE VERIFICATION FOR JAVA

In [1], we presented an extension to deductive verification of Java programs that allowed us to automatically compute verification coverage. The tool is based on standard deductive verification techniques. It analyzes a program given as Java bytecode one procedure at a time. Each procedure is first translated into the intermediate verification language Boogie [3]. However, unlike in weakest-precondition-based deductive verification, run-time assertions are not modeled as assertion statements but as conditional choices where a violation of the assertion actually creates a new exception and returns. For each such exception, we keep a mapping of the Java bytecode instruction that it was generated for.

Now, we create a first-order logic formula representing the transition relation of this procedure. That is, a model of this formula can be mapped to a feasible execution of the procedure. Further, for each model, we can extract a blocking clause that blocks all models that correspond to executions of the same control-flow path. That is, by iteratively checking the formula and blocking feasible paths, we can obtain a control-flow graph cover of all feasible paths.

We use this formula to first identify all control-flow paths that are feasible under the postcondition that no run-time exception is thrown. Then, we lift this postcondition, to cover all paths that may throw runtime exceptions. Everything that is not covered in either of the steps is unreachable. That is, this algorithm allows us to identify which statements in the Boogie program are reachable, and which are only reachable on executions that throw run-time exception. Using this information and the mapping from Boogie statements to Java bytecode instructions, we can decide for each bytecode instruction if it may, may-not, or must throw a run-time exception, or if it is unreachable. Hence, we can simply compute the verification coverage from the sum of instructions that may or must throw run-time exceptions, divided by the overall number of instructions.

Now the question is, how sound and complete can we implement this way of computing verification coverage? For Java, we have the benefit that we can use the relatively simple Burstall-Bornat memory model (e.g., [5]), which allows us a sound and relatively precise modular analysis. As most run-time assertions in the bytecode are related to whether or not a variable points to an allocated object, a very coarse (but sound) abstraction is sufficient in most cases (see [2] for a detailed description of the encoding). Our approach inevitably loses soundness in the presence of threads or reflection. For now, we do not see any feasible way to extend our tool to handle these language features properly. Completeness is always a trade-off between precision and cost. In our current implementation, for example, we do not try to analyze loops or function calls. We simply replace them by the weakest possible contract (i.e., a non-deterministic assignment to all variables that may be modified followed by the assertion of the trivial invariant *true*).

Table 1 shows our preliminary results of computing verification coverage for some open-source applications. The source code for all applications was taken from recent stable releases. Our algorithm to compute verification coverage was applied to each procedure of each program with a timeout of 10 seconds. Note that the analysis is *not* inter-procedural. Each procedure is analyzed without any precondition.

Completeness. The results show that a relatively high verification coverage of up to 80% can be computed by our tool. Following popular defect metrics for defect density (e.g., one defect per thousand lines of code in open-source software), this is certainly relatively low, but given that the analysis is not inter-procedural and loops are abstracted in a brutal way, it should be possible to drive this percentage up relatively quickly with simple preconditions and invariants.

Soundness. The approach is inevitably unsound because it does not handle parallelism or reflection, which are both frequently used in real code. We checked the soundness of the approach by manually expecting roughly one hundred statements. For all statements that were reported *strictly unsafe* (i.e., that must throw an exception), we could find a source of unsoundness, mostly related to multi-threading. The same could be observed for over half of the unreachable statements. In most cases, code was reported unreachable because it was preceded by a loop that only terminated if another thread set a flag. We did not find any statement that was wrongly reported to be safe. However, given the small sample, this result is more of an intuition than a reliable trend.

Future Work. So far, we have seen that verification coverage can be computed on top of deductive verification for Java. The next step has to be to find experimental evidence that verification coverage correlates with code quality. To that end, we plan two sets of experiments: in the first experiment, we will take some open-source projects that are relatively mature, where source code changes in the repository were mostly made to fix bugs and improve quality (rather than adding new functionality). For the projects we will compute verification coverage over several source code versions to see if quality improvements can be captured by our metric. In a second experiment we will try to verify a piece

Program	*#procedures*	*#stmts*	*#unreachable*	*#safe*	*#strictly unsafe*	*#possibly unsafe*	*#skipped*
ArgoUML	13,515	142,959	31	110,920 (75%)	2	20,125	11,881
Args4j	361	2,489	0	2,011 (80%)	0	311	167
GraVy	2,044	31,860	6	16,522 (51%)	0	3,844	11,488
Hadoop	18,728	266,571	54	177,373 (66%)	7	32,249	56,888
Log4j	3,172	30,611	1	22,381 (73%)	0	2,746	5,483

Table 1: Results of computing verification coverage for several pieces of software with a timeout after 10 seconds. The last column represents that number of statements that have not been analyzed because their containing method reached a timeout.

of software and measure if verification coverage improves as we proceed with our verification efforts.

An important aspect of our future work has to be the assessment of unsoundness. While the approach as is, is unsound because of its lacking support for multi-threading and reflection, it is very important to keep in mind that any intermediate result of deductive verification is inevitably unsound as long as there are preconditions which are not yet verified. That is, while verification researchers tend to be very religious about the idea of soundness, a very different approach to this is required when trying to measure verification progress. Unsoundness is inevitable when measuring progress and we are looking for ways to incorporate this in our metrics.

3. ABSTRACT INTERPRETATION FOR C

In a second experiment we tried to compute the verification coverage of the domain name server BIND using the abstract interpretation based analysis from Frama-C [4]. Intuitively, abstract interpretation is the better choice to compute verification coverage. Abstract interpretation keeps a symbolic state for the program while it iterates through the program. Every time it reaches an (implicit) assertion, it can check if this assertion may fail when executed from the current symbolic state. The symbolic state may become imprecise through widening operations, when tracking all possible precise states becomes infeasible within the given memory constraints.

For our experiments, we first used Frama-C in order to convert each implicit assertion into an explicit one: it generated a total of 46726 assertions. Then we applied Value in a modular way by starting the analysis from each function of BIND. It results than 8122 (17%) of these assertions have been verified by Value, while 11234 assertions (24%) hav not been verified, and the remaining 27368 (59%) were not reached by Value because our methodology is not complete (see below).

That is, the results are somewhat discouraging as compared to what could be achieved for the Java benchmarks. The main reason for this seems to be the more complex memory model that is needed when analyzing `C` programs. Unlike Java, `C` can have very complex aliasing. Therefore, when analyzing a procedure in isolation, one either has to make strong assumptions about the state of the memory upon entering the procedure or start from a very imprecise initial state. In our experiments, we decided to start from the imprecise initial state as we tried to avoid creating proof-obligations that we are not going to check.

Completeness. Like in the previous experiment, our results are very incomplete. It turned out that applying Frama-Cs Value analysis in a modular way is not very suitable. Currently, the tool does not provide adequate support to infer a suitable initial state for analyzing procedures in isolation. In particular the high number of unreachable assertions indicates that additional work is required to make this approach feasible.

Certainly these problems do not appear if abstract interpretation is started from a genuine entry point of the program. However, in that case, the analysis starts to lose precision rather quickly and fails to progress deeper into the control-flow graph without human guidance.

Soundness. While Frama-C is sound in general, some of the assumptions that we had to make in order to do a modular analysis were unsound. As the overall result of the experiment already shows that this approach is not suitable to compute verification coverage, we did not further investigate how unsoundness affects the results.

Future Work. Even though the results of this experiment are rather discouraging, abstract interpretation still seems to be the more intuitive way of computing verification coverage. In the future, we will investigate if there are more suitable ways of doing modular abstract interpretation (e.g., [6]), or if the problem is only rooted in the more complex memory model that is needed to analyze `C` programs.

4. CONCLUSION

Our work presents a first step in our effort to quantify the software quality improvements that can be achieved by using formal verification. We believe that the concept of verification coverage is useful in general, and our experiments for Java programs already look promising. For our future work, we have to carry out several case studies to validate that verification coverage is a robust metric.

5. REFERENCES

[1] S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar. The Gradual Verifier. In *NASA Formal Methods*, pages 313–327, 2014.

[2] S. Arlt, P. Rümmer, and M. Schäf. Joogie: From java through jimple to boogie. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 3–8, New York, NY, USA, 2013. ACM.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, pages 364–387, 2005.

[4] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - A Software Analysis Perspective. In *SEFM*, pages 233–247, 2012.

[5] K. R. M. Leino and P. Rümmer. A Polymorphic Intermediate Verification Language: Design and Logical Encoding. In *TACAS*, pages 312–327, 2010.

[6] F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI*, pages 283–298, 2007.