Survey of Software verification for real world applications

Candidate No: 105936

16th January 2019

Contents

1 Introduction

Generally software verification is a very interesting topic in research at the moment, however it is currently limited to the field of researchers and it it is only really used as a part of demonstration software and only as a part of verifiable languages. Therefore this paper will look into how the software verification techniques can be applied to applications designed for use in the real world. This will obviously have advantages as it guarantees code free of fatal runtime errors and reduces the likelihood of other errors.

2 Current Problem

Currently if we compare software verification to unit testing as a form of guarantee against bugs we see that unit testing is far more popular and there are far more frameworks available to use for unit testing than there are for verifying software. Also, if you look at the current Computer Science Degree at the University of Sussex unit testing is taught in the first year and software verification is not taught until the masters level. This paper will look into the different software verification techniques and why they are not used more often.

3 Software Verification Overview

Software verification can be looked at as the automatic analysis of a program. One commonplace example of program verification is type checking which has been implemented in a number of programming languages. There are also more complex areas of program verification such as extended static checking and full functional program verification. For the purposes of this report I will be calling Languages which have been designed to be verified as a part of compilation "Verifiable Languages". There are a number of these languages which I will go on to outline further in the paper. There are also tools which have been develop to verify existing languages

4 Existing Software verification techniques

There are three types of static analysis are Abstract static analysis, model checking and, bounded model checking.[1]

4.1 Abstract Static Analysis/Interpretation

This verification technique was introduced in [2] as a way of reducing runtime errors, they saw that strong typing was a start in reducing run time errors and then went on to look into how to make pointers safer. Static analysis allows for the analysis of a program without actually executing the program. The way in which it works is by computing a superset of possible values for each stage of the program. You can then look at the sets for example if one of the set of values for a divisor is zero then you may have a divide by zero error. Obviously if these are not in the set of values you can guarantee that the program does not contain divide by zero errors.

4.2 Model Checking

This involves looking at a program as a set of states and transitions an algorithm can then check the reachable states of the program an find if there is a case where the program may not terminate[1]. It is possible to provide properties to clarify and restrict variables as explained in the following quote. "In general, properties are classified to 'safety' and 'liveness' properties. While the former declares what should not happen (or equivalently, what should always happen), the latter declares what should eventually happen."[3] which allow you to show that bad states are inaccessible.

4.2.1 Bounded Model Checking

This was introduced in a 1999 paper by Biere et al.[4]. As an effort to reduce the complexity, it is a development on Model checking due to limited computing power and the growing complexity of programs it became difficult to run model checking on programs. Therefore, BMC allows for checking with a limited number of steps.

5 Existing Verification Tools

As a part of looking into how to include software verification in real world applications we need to look at how they can be used in mainstream languages therefore in the following sections I have described a number of tools and what features they support in verifying programs written in mainstream languages.

5.1 Extended Static Checker for Java[5]

This is a checker which finds common programming errors, this may be one of the problems with these tools they are labelled as tools which find bugs rather than tools which guarantee code quality if you are a programmer generally you don't like finding bugs in your code. Programmers need to write annotations for the program. "The checker is powered by verification-condition generation and automatic theorem proving techniques."[5]. This works on a modular level meaning any method or constructor can be verified.

5.2 Static Driver Verifier/SLAM[6]

This was designed to allow Microsoft to verify drivers for their operating systems. The actual tool used to verify drivers was called Static Driver Verifier while slam is the analysis engine it uses. SLAM is actually a tool which can check if a C program correctly uses an external library. I expect that this is one of the best examples of mainstream software verification uses. This is probably because the SDV(Static Driver Verifier) tool is included as a part of the driver developer kit for windows this means that developers have to make sure that their driver is verified before they can release it.

5.3 Spec#

This is not a tool as such it is a language which adds various static analysis features which are checked during compilation. This runs on the .NET framework therefore can be used with other .NET languages. Many developers have easy access to the language through visual studio "The Spec# system is fully integrated into the Microsoft Visual Studio environment."[7]. Barnett et al. also lists a number of features which Spec# adds to C# such as:

- type support for distinguishing non-null object references from possibly-null object references
- method specifications like pre- and postconditions
- support for constraining the data fields of objects

Because this works on the .NET framework functions can be called from other languages meaning it is possible to call a method with parameters which violate the preconditions in this case Spec# provides the option to specify an exception to be thrown in the case where a precondition is not met.

5.4 Spark Ada

Spark is a Verifiable language based on Ada it uses pre and post conditions as contracts for verification. It appears to be used on a number of mission critical pieces of software mainly on embedded systems.

5.5 Krakatoa [8], [9]

This is a verification tool which runs can be run on programs written in Java. Contracts for the code are written in the comments for a method the come from the Java Modelling Language[10]. This tool is integrated into the Why IDE which was designed to make it easier to verify programs.

6 Current State of software verification in Mainstream

6.1 Software Verification In Education

Software verification is not really taught in Universities as mentioned earlier however there is an interesting paper from Feinerer et al. [11] which talks about choosing a a tool to use for teaching Formal software verification. The paper states that "we think that automated program verification has become mature enough to spread the word in industry, via the students."[11] bear in mind that this was around 10 years ago therefore it is still surprising why formal software verification hasn't taken off.

6.1.1 Software Quality in education

In my experience software quality tends to be very low in education this is mainly due to the fact that the project will usually be thrown away or archived after it has been submitted. The software is also usually only tested very gently therefore having formal verification to add code quality seems like a waste of time to students as they're only concerned with functionality of the designed application as that is what is generally tested.

6.2 Software Verification In Industry

Perhaps unsurprisingly software verification seems to be favoured in industries where there is a need for bug free mission critical software for example the space and aviation industry[12], [13] and in large software companies for some of their more mission critical projects[14] I.E. compliers and Operating system kernels.

7 Ideas on how to improve Software verification in the Mainstream

7.1 Introduce a verifiable language which compiles to the JVM

Ideally this would work similar to the Spec# language where it would simply be Java with some extras which makes it verifiable with it also compiling to the JVM so that it can be used in conjunction with other JVM libraries. It would also need to be integrated into an IDE or have various IDE plug-ins to make it easier to start developing using it.

7.2 Introduce software verification in education

From some very quick research I can see that Formal Software verification is actually taught at a handful of universities. But most of these are focusing on teaching how to formally verify software rather than how to use verification tools. An example of this is the Krakatoa tool mentioned in section 5.5 which is used by the University of La Rioja in their teaching of formal verification as it runs through the steps of the proofs[15]. Obviously understanding how it works is important but I would argue that for developers at least, understanding how to use the tools is an earlier step which should be taken. This would involve giving formal software verification a similar amount of time to what unit testing currently gets thereby including it in all undergraduate degrees. One of the barriers which may be stopping this is the logic behind the automatic verification which is very complex and difficult to understand although the logic does not necessarily need to be taught to understand how to use formal software verification.

8 Conclusion

In conclusion there seems to be many tools and many verifiable languages although there doesn't appear to be any which compile to the JVM which could be the reason why it is not more widespread. I would expect formal verification to grow in the future but as mentioned earlier there a number of people already predicted this and this obviously has not happened.

References

[1] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008, ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410.

[2] P. Cousot and R. Cousot, "Static determination of dynamic properties of generalized type unions," *SIGPLAN Not.*, vol. 12, no. 3, pp. 77–94, Mar. 1977, ISSN: 0362-1340. DOI: 10.1145/390017.808314. [Online]. Available: http://doi.acm.org/10.1145/390017.808314.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," in *Advances in Computers*, Elsevier, 2003, pp. 117–148. DOI: 10.1016/s0065-2458(03)58003-2.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Jan. 1, 1999, ISBN: 978-3-540-65703-3. DOI: 10.1007/3-540-49059-0_14. [Online]. Available: http://dx.doi.org/10.1007/3-540-49059-0_14.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02, Berlin, Germany: ACM, 2002, pp. 234–245, ISBN: 1-58113-463-0. DOI: 10.1145/512529.512558. [Online]. Available: http://doi.acm.org/10.1145/512529.512558.

[6] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft," in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2004, pp. 1–20. DOI: 10.1007/978-3-540-24756-2_1. [Online]. Available: https://www.microsoft.com/en-us/research/publication/slam-and-static-driver-verifier-technology-transfer-of-formal-methods-inside-microsoft/.

[7] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Springer Berlin Heidelberg, 2005, pp. 49–69. DOI: 10.1007/978-3-540-30569-9_3.

[8] C. Marché, *The Krakatoa Verification Tool for JAVA programs*, Version 2.41, INRIA Team Toccata, Batiment 650, Université Paris-Sud 91405 Orsay cedex, France, Jun. 2018. [Online]. Available: krakatoa.lri.fr/krakatoa.pdf.

[9] J.-C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus platform for deductive program verification," in *Computer Aided Verification*, Springer Berlin Heidelberg, 2007, pp. 173–177. DOI: 10.1007/978-3-540-73368-3_21.

[10] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, Dec. 2004. DOI: 10.1007/s10009-004-0167-4.

[11] I. Feinerer and G. Salzer, "A comparison of tools for teaching formal software verification," *Formal Aspects of Computing*, vol. 21, no. 3, pp. 293–301, Jun. 2008. DOI: 10.1007/s00165-008-0084-5.

[12] R. Feldt, R. Torkar, E. Ahmad, and B. Raza, "Challenges with software verification and validation activities in the space industry," in *2010 Third International Conference on Software Testing, Verification and Validation*, IEEE, 2010. DOI: 10.1109/icst.2010.37.

[13] S. D. Nelson and C. Pecheur, "Formal verification for a next-generation space shuttle," in *Formal Approaches to Agent-Based Systems*, Springer Berlin Heidelberg, 2003, pp. 53–67. DOI: 10.1007/978-3-540-45133-4_5.

[14] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill, "An overview of the Singularity project," Microsoft, Microsoft Research One Microsoft Way Redmond, WA 98052, Tech. Rep., Oct. 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/an-overview-of-the-singularity-project/.

[15] A. Romero and J. Divasón, "Experiences and new alternatives for teaching formal verification of Java programs," in *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE 2018*, ACM Press, 2018. DOI: 10.1145/3197091.3205811.