Survey of formal program verification tools and languages

Candidate No: 105936

17th January 2019

Contents

1 Introduction

This paper is primarily aimed at providing an introduction over some software verification techniques and tools. More information on each of these can be found by following the references. The paper will firstly start with providing a high level overview of the Topic of software verification before going into some of the techniques used for software verification after which several tools and verifiable languages will be listed and features will be described.

2 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 many 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 developed to verify existing languages

3 Existing Software verification techniques

According to D'Sliva et al. there are three types of static analysis are Abstract static analysis, model checking and, bounded model checking[1].

3.1 Abstract Static Analysis/Interpretation

This verification technique was introduced in [2] by Cousot and Cousot as a way of reducing runtime errors, they saw that strong typing was beneficial in reducing run time errors and then went on to investigate how to make pointers safer. Static analysis allows for the analysis of a program without 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.

3.2 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.

3.2.1 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.

4 Formal Program Verification Tools

4.1 Extended Static Checker for Java[5]

4.1.1 Background

This tool was developed by a group at Compaq Systems. And is based on a section of formal program verification Flanagan et al. called “Extended Static Checking”[5]. and builds on theory behind static type checkers.

4.1.2 Details

This works on a modular level meaning any method or constructor can be verified. Although for this to be implemented the users need to add annotations to the methods so that they can be checked against a specification. The paper[5] styles these as equivalent to comments on the input and outputs of a method.

4.1.3 Examples of usages

The authors of Mercator used ESC/Java on their project they were able to annotate 7000 lines of code in around 6 hours and it did find a bug in the project. The developers of ESC/Java also ran the tool on the front end of ESC/Java, it took around 3 weeks to annotate the 30.000 lines of code but found around half a dozen errors[5] the developer decided that the errors found were not worth the three weeks work. However, later on one of the developers conducted a major change in the system during which a bug was introduced. The system would pass all of the testing suites but when trying to run the program it would crash. It took 2 hours to find the problem whereas running ESC/Java on the entire program only took 73 minutes and 3 for the file which contained the error. This shows that it is still beneficial to use this tool even with the extra overhead of adding the annotations due to the advantages gained from greater maintainability.

4.2 Static Driver Verifier/SLAM[6]

4.2.1 Background

Due to the problem of device drivers needing to run on the operating system kernel, and therefore poorly written or malicious device drivers causing full OS level crashes. Microsoft decided to produce a tool which "could automatically check that a C program correctly uses the interface to an external library"[6].

4.3 Details

The actual tool used to verify drivers was called Static Driver Verifier while SLAM is the analysis engine it uses. 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.

4.4 Krakatoa [7], [8]

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[9]. The Why tool allows users of krakatoa to follow the proofs for their program making it a powerful teaching tool as an example Romero et al. uses it as a part of their teaching at the University of La Rioja[10].

4.5 KeY

The KeY project is about trying to verify Java programs. There are some features which KeY does not currently support such as Generics and Lambdas[11].

4.5.1 Examples of Usages

This has been used to verify a part of a Electronic voting system written in Java and KeY has also been used to verify a couple of sorting algorithms

4.6 Java PathFinder

4.6.1 Background

Pathfinder was originally developed as a state model checker but has since been developed to use many verification techniques[12].

4.6.2 Details

This analyses the Java bytecode this means that it can be run on any language which compiles to Java bytecode such as Scala or Kotlin.

5 Verifiable Languages

5.1 Spec#

Barnett et al. describes this as a programming system, it is designed as a small extension to C# 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."[13]. Barnett et al. also lists several features which Spec# adds to C# such as:

- "type support for distinguishing non-null object references from possibly-null object references"[13]
- "method specifications like pre- and postconditions"[13]
- "support for constraining the data fields of objects"[13]

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.

5.2 Dafny

This is another language from Microsoft, It has been developed to be easy to use for non experts in the area of software verification and avoid the need to understand the formal proofs like the tool in section 4.4. Error messages are calculated in the background while the developer is working.[14] When a program written in Dafny is verified it is translated into Boogie which is a verification language. This can then be verified which proves the verification of the original Dafny program[15].

5.3 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. This is actually quite a old "Verifiable Language" with there being a few versions of spark with the first being base on Ada 83.

5.4 Boogie

This is an open source tool which uses the Z3 SMT solver to verify the Boogie language. It was developed with Spec#. There are a number of languages which have tools which allow them to be translated to Boogie an then verified. The language is made up of[16]:

- Type
- Constants
- Functions
- Axioms
- Global Variables
- Procedure Declarations
- Procedure Implementation

These are then turned into logical formulas which can then be formally verified.

6 Other Tools/Languages

6.1 JML - Java Modelling Language

6.1.1 Background

This was designed to provide documentation for interfaces and allow for the communication of the behaviour of programs.

6.2 Details

JML is a type of BISL(Behavioural Interface Specification) this means that is a language which describes a programs interface and behaviour. JML can either be in their own files or as annotations in Java[17].

6.3 Examples of usages

As mentioned in section 4.1 JML has been used to provide specifications/contracts as a part of their bug finding tool. It has also been used in the University of Nijmegen as a part of their tool to provide total correctness checking to a program[18].

7 Conclusion

In Conclusion there are many practical applications of formal program verification techniques which could be used. Some of which focus on adding formal verification to existing programs and some which focuses on writing new formally verified programs or rewriting programs in verifiable languages. Given more time I would have liked to conducted more research into many of the tools and languages. Also, I would have liked to conduct some experiments with how easy each of the tools and languages are to use and if they all verify the same program correctly.

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] 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`.

[8] 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`.

[9] 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.

[10] 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.

[11] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification – The KeY Book*. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-49812-6.

[12] P. Mehlitz, N. Rungta, and W. Visser, “A hands-on Java pathfinder tutorial,” in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, May 2013. DOI: 10.1109/icse.2013.6606756.

[13] 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.

[14] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer Berlin Heidelberg, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

[15] R. Leino, “Specification and verification of object-oriented software,” Jun. 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/specification-verification-object-oriented-software/.

[16] K. R. M. Leino, “This is boogie 2,” Jun. 2008, [Online]. Available: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[17] G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary design of JML,” *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, p. 1, May 2006. DOI: 10.1145/1127878.1127884.

[18] B. Jacobs and E. Poll, “A logic for the Java modeling language JML,” in *Fundamental Approaches to Software Engineering*, Springer Berlin Heidelberg, 2001, pp. 284–299. DOI: 10.1007/3-540-45314-8_21.