

2019A7PS0020U

by Dhruv Maheshwari

Submission date: 26-Dec-2020 09:30AM (UTC+0400)

Submission ID: 1481286966

File name: 2019A7PS0020U.docx (361.07K)

Word count: 2216


Character count: 11989



DECEMBER 25, 2020

LOGIC ASSIGNMENT - 2

KSHITIZ BANSAL 2019A7PS0012U
DHRUV MAHESHWARI 2019A7PS0020U
RANVIRJIT SINGH AHLUWALIA 2019A7PS0027U
VIGNESH BHAT 2019A7PS0088U



Acknowledgement

The Program Verification project has been a great learning for the team. It has built a foundation for us to develop more complex programs. Over the duration of project, the team members went through endless debates, arguments and white boarding sessions. This experience has not only made us more knowledgeable but also brought us closer as friends. We faced challenges due to Covid restrictions but due to our relentless pursuit, we were to organize the inputs in time. We would like to thank our teachers for their encouragement, support and invaluable insights without whom we would not have completed the project. We are also grateful to resources like our library, the internet and seniors of our university from where we were able to piece the nuggets to create this wonderful program. We humbly acknowledge all the help and support received in helping us achieve success with the Program Verification project.

Index

<u>Sr No</u>	<u>Topic</u>	<u>Page no</u>
1	History of Program Verification Techniques	3
2	Significance of Program Verification	7
3	Tools and Techniques	8
4	Program Verification Methodology	10

History of Program Verification Techniques

With the advent of design complexity, the conventional simulation-based verification could not yield satisfactory results. As a result, many new techniques started originating.

¹ In the year 1954, the first computer-generated mathematical proof for a theorem for a decidable fragment of first order logic called Pressburger arithmetic was developed by Martin Davis. In April 1970, the design community was challenged by Edsger Wybe Dijkstra who said, “Testing shows the presence, not the absence of bugs”. This led to many thinkers moot the idea for formal/program verification.

Historical Origins

Edsger W Dijkstra



“Program testing can be used to show the presence of bugs, but never to show their absence!”

Some interesting quotes from him [[Source: EWD303](#)]

I shall argue that our programs should be correct

I shall argue that debugging is an inadequate means for achieving that goal and that we must prove the correctness of programs

I shall argue that we must tailor our programs to the proof requirements

I shall argue that programming will become more and more an activity of mathematical nature

Theorem proving

¹ To validate the computer programs written in late 1960s, first order theorem provers were applied. One example was Stanford Pascal Verifier.

¹ The first machine-based prover, Nqthm was developed in Edinburgh, Scotland in 1972 by Robert S. Boyer and J. Strother Moore. It further paved way for A Computational Logic for Applicative Common Lisp (ACL2) which was able to solve the proof for mathematical theorems.

Logic for Computable Functions (LCF) systems were built for proof checking by Sir Robert Milner. It laid the foundation for advanced systems such as HOL 4, HOL Light, Coq etc.

These systems were used for the verification of floating-point numbers in digital design extensively by companies such as Intel and AMD.

Theorem Provers

Machines that build proofs using inference rules and decision procedures

Powerful tools, no capacity limits, but no bug finding, and training can be a challenge!



We have come a long way but the user base is still quite small...

Fascinating book by Freek Wiedijk: The Seventeen Provers of the World, 2008, Springer Verlag

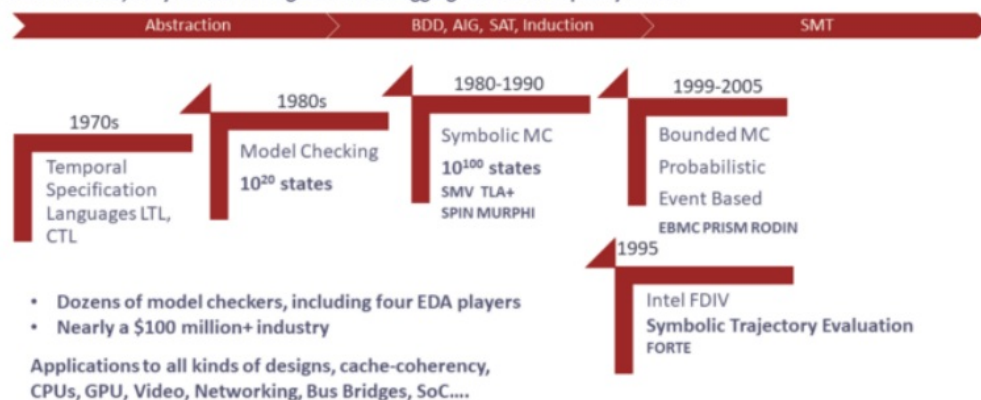
Model Checking

As Theorem checking was not that effective in many cases (except for infinite systems), a need was felt for a better program verification technique. This problem started getting addressed by many people in the 1970s and 1980s.

Model Checkers

Bug hunting machines that build proofs through exhaustive state-space search

Powerful tools, easy to learn and great for debugging but suffer capacity limits!



Useful reference: Ed Clarke et al. Handbook of Model Checking, 2018

Finally, in the year 1981 a breakthrough was achieved as the first automated model checking algorithm was developed. The pioneers were Emerson and Clarke, who combined the state-exploration approach with temporal logic. This was a very effective model; it was fast, accurate, could handle verify the programs and give counter examples when it could prove!

Soon after this achievement, people felt to apply verification techniques on hardware too. However, there was a serious concern as the above algorithm had limited state-space reachability. With this technology, we could only verify 100s of states. With the advancement in technology, we can today verify digital designs as large as 10^{120} states!

Randall Bryant began to take part in getting involved with the proposal of simulation at the circuit-level for the simulation of logic in transistors. Especially, he thought of utilizing the suggestion of a simulation which was three-valued. Randall also searched the application of encoding which was symbolic in the compressing of simulation test vectors. The biggest roadblock was an efficiency of encoding the symbols and also the Boolean formulas which were made based on them. Randall made ordered binary decision diagrams (OBDDs). Ken McMillan on working with Edmund Clarke, came to know about this and symbolic model checking was created in 1993.

A canonical form was provided by OBDD's for Boolean formulas which are more compact than the disjunctive or normal conjunctive forms. Efficient algorithms have been created for manipulating them.

Rendall and Carl J. Seger created one more very strong model-checking technique called symbolic trajectory evaluation (STE), which has been utilized a lot for processor verification by many companies. Pentium floating-point units have been proved using symbolic trajectory evaluation, and Intel still uses this technique.

STE employs three-value logic (0, 1, X) with partial order over a lattice to carry out symbolic simulation over finite time periods. The "X" denotes an unknown state in the design, along with the ability to perform Boolean operations on "X" values (e.g., $0 \text{ AND } X = 0$, $1 \text{ AND } X = X$), supported an automatic fast data abstraction. This along with BDDs provides a very fast algorithm for finite-state model checking.

The main disadvantage of STE is that it is unable to specify properties over unbounded time periods. This limits its applicability to verifying digital designs for finite bounded time behavior.

Other prominent developments in formal technology still in use that include model checking and some form of theorem proving for the verification of concurrent systems include the following:

- The TLA+-based model checker by Leslie Lamport
- The CSP-based verification by Tony Hoare
- The Alloy language from Daniel Jackson at MIT
- The Event-B language, Z-notation, and proof tools from Jean-Raymond Abrial

Equivalence Checking

In order to aid the above two methods, a third method came into picture. This helps more in verifying hardware.

Equivalence Checkers

Bug hunting machines that establish proofs through exhaustive comparison of two models

Powerful technology, can have capacity limits, but debug and learning is easy!

- Equivalence checking (EC) is well known for synthesis verification
- At the core of EC we need at least two models to perform comparison
- The comparison can be either:
 - Combinational [identical state encoding]
 - Sequential [different state encoding]
- Most sequential EC in practice is conditional
- Half a dozen commercial EC tools exist today

This method takes into account two design models. It compares them and tells either they are equal or gives a counter example to prove when they aren't same. Early forms of equivalence checking were targeted at combinational hardware designs. Scalable equivalence checkers were developed for the equivalence checking of sequential designs.

11

Hardware designers use equivalence checkers to compare an unoptimized design with its optimized counterpart.

Significance of Program Verification:

Computers are frequently used in systems which are needed to check for safety where a small bug could cause loss of life. These bugs can be checked by formal verification for example in telephone exchanges, heart pacemakers, radiation therapy machines, aircraft, car engine management systems and nuclear reactor controllers.

Bugs can cause financial problems if a product is faulty. For example, in 1994 a bug called FDIV which was present in the Intel Pentium processor cost them 500 million.

Nowadays new products are raised at an exponential rate. So, this has sparked an interest in Intel to find techniques to reduce errors which can be done using program verification.

A good approach is to have the check the proof using a computer program. It can decrease the mistakes and automate some sections of proofs. Detailed human guidance is required as there are restrictions on the getting it automated

Many problems can be solved using decision methods with limited human intervention, basically by using Boolean equivalence checking or Temporal logic model checking or Symbolic trajectory evaluation, this makes formal verification successful in hardware.

Formal verification is useful in proving the correctness of cryptographic methods, combinational circuits, digital circuits with internal memory.

The increase in complex design has increased the significance of formal verification techniques in the hardware industry. Formal verification is failing to progress in the software industry but it is heavily utilized by most hardware companies. This could be because of its need in the hardware industry, where errors have more commercial weightage. Because of the potential fine-drawn interactions between the different components, it is harder to utilize many number of possibilities by simulation. Significant features of hardware design are manageable to automated proof techniques, making formal verification easier to be more productive.

Many operating systems have been verified formally since 2011 for example NICTA's Secure Embedded 14 microkernel which was sold commercially as seL4 by OK Labs and OSEK/VDX based real-time operating system ORIENTAIS by Green Hills Software's Integrity Operating System, SYSGO's PikeOS and East China Normal University.

Yale University professors in 2016 created a formal verification protocol for blockchain which they called CertiKOS, which is the being used as a security program.

Formal verification has been applied to the design of big computer networks through a mathematical model of the network in 2017, and as part of a new network technology category, intent-based networking. Some network software vendors offering formal verification solutions are Cisco Forward Networks and Veriflow Systems.

A formally verified C compiler implementing the majority of ISO C is theCompCert C Compiler.

Tools and Techniques:

There are many tools and techniques that are used for program verification. Two main methods of program verification we will discuss are Model Checkers and Static Program Analysis:

2 Model Checking:

Model checking is a technique that is dependent on building a finite model of a required system and checking if it holds the required property. With advancement of algorithmic techniques like partial order reduction, compositionality and Binary Decision Diagrams allows for automatic and exhaustive analysis of finite state models.

Some tools used for model checking are:

1. Finite-state Model Checkers:

a. VisualSTATE :

This commercial tool is used for supporting code generation from hierarchical state machine models compliant with UML standard. The tool also offer full verification of a number of generic sanity properties (e.g. absence of deadlock) and simulation capabilities.

Visual state is a tool used to generate code form hierarchical state machine models complaint with the UML standard. The tool also offers full verification of a number of generic sanity properties and simulation capabilities.

b. FormalCheck:

FormalCheck is another of the common tools used for program verification of complex control units by the use of several different reduction techniques that supply us with sophisticated methods for dealing with the verification of large circuits.

2. Model Checkers Based on Process Algebra:

a. Ceaser/Aldebaran:

The Ceaser tool suite is built around the LOTOS process algebra. It is used in various kinds of equivalence checking, visualization tools, model checking tools and simulation tools. It is maintained by VASY/INRIA.

b. muCRL:

The muCRL tool suite is built around the muCRL and timed muCRL process algebra. It uses tools and tehniques that operate on a symbolic representation of a state system which incudes unbounded data types. Such representation is used by other tools to simulate behaviors, verify modal mu-calculus formulae and check for equivalence.

²3. Real-Time and Hybrid Model Checkers:

a. ⁸Kronos:

Kronos is a tool built around real-time systems created as ⁸timed automata and correctness expressed in real-time temporal logic TCTL, which is an extension of CTL and allows temporal reasoning over time.

b. ⁶UPPAAL:

UPPAL is a tool used for modelling, validation and verification of real-time systems modelled as networks of timed automata expanded with discrete data types.

4. Stochastic Model Checkers:

a. ETMCC:

ETMCC is a model checker for stochastic systems using the PCTL logic, an extension of CTL and associated model checking algorithms.

b. ²Rapture:

Rapture is a tool ²designed to verify reachability of Markov Decision Processes.

5. Model Checking for Source Code:

a. ²Bandera:

Bandera is a tool suite designed to bridge the semantic gap between a non-finite state system expressed a source code and preferred input format for existing model checkers.

b. ²BLAST:

BLAST is a software checker used to check programs in C by using counter example-²driven automatic abstraction refinement to construct an abstract model which is checked for safety properties.

²Static Program Analysis:

Static program analysis executes an abstract version of a program's semantics on abstract data instead of concrete data. Abstraction and concretization functions exist between the two domains mapping (sets of) concrete data to their most precise description and mapping abstract data to the set of represented concrete data. The abstract semantics of the program statements is applied iteratively until a fixed point is reached. This fixed point describes properties of all program executions at each program point.

The tools used for this are:

- a. ²PolySpace Verifier is a general purpose tool for analyzing C and Ada programs for run-time errors.
- b. The Program Analyzer Generator, PAG, is a tool supporting the automatic generation of program analyses from specifications.
- c. BANE is research tool for experimentation with program analyses.
- d. The aiT WCET analyzers of AbsInt determine bounds on execution times by abstract interpretation.

Program Verification Methodology

We are going to take a program to find the nth Fibonacci number in java and prove it using Hoare's Logic by using proof tableaux using partial correctness:

Java Code:

```
// Fibo.java

class Fibo {
    5 public static void main(String args[]) {
        int a = 0;
        int b = 0;
        int c = 1;
        int i = 1;
        int n = 7;
        while(i!=n) {
            a = b;
            b = c;
            c = a + b;
            i = i+1;
        }
        System.out.println(c);
    }
}
```

Program Code:

On stripping the java code down to the program code:

```
a = 0
b = 0
c = 1
i = 1
n = 7
while(i!=n) {
    a = b
```

```

12
b = c
c = a + b
i = i + 1
}

```

Proof Tableaux:

(T)

a = 0

b = 0

$$\langle 1 = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}} \rangle$$

Implied

c = 1

$$\langle c = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}} \rangle$$

Assignment

i = 1

n = 7

$$\langle \forall 0 < i < n, c = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}} \rangle$$

Assignment

while(i!=n) {

$$\langle \forall 0 < i < n, c = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}} \wedge (i \neq n) \rangle$$

Inv. Hyp. \wedge guard

$$\langle \forall 0 < i < n, (b + c) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i+1}}{\sqrt{5}} \rangle$$

Implied

a = b

$$\langle \forall 0 < i < n, (a + c) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i+1}}{\sqrt{5}} \rangle$$

Assignment

b = c

$$\langle \forall 0 < i < n, (a + b) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i+1}}{\sqrt{5}} \rangle$$

Assignment

c = a + b

$$(\forall 0 < i < n, c = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i+1}}{\sqrt{5}})$$

Assignment

i = i + 1

$$(\forall 0 < i < n, c = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}})$$

Assignment

}

$$(\forall 0 < i < n, c = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}} \wedge \neg(i \neq n))$$

Partial-while

$$(c = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}})$$

Implied

ORIGINALITY REPORT

35%

SIMILARITY INDEX

27%

INTERNET SOURCES

12%

PUBLICATIONS

3%

STUDENT PAPERS

PRIMARY SOURCES

1

www.eeweb.com

Internet Source

14%

2

Bruno Bouyssounouse, Joseph Sifakis.
"Embedded Systems Design", Springer Nature,
2005

Publication

8%

3

en.wikipedia.org

Internet Source

7%

4

www.cl.cam.ac.uk

Internet Source

2%

5

how2j.cn

Internet Source

1%

6

www.inf.brad.ac.uk

Internet Source

1%

7

Bruno Bouyssounouse, Joseph Sifakis.
"Chapter 7 Tools for Verification and Validation",
Springer Science and Business Media LLC,
2005

Publication

1%

8	Madhusudanan. J, Anand. P, Hariharan. S, V. Prasanna Venkatesan. "Verification of Generic Ubiquitous Middleware for Smart Home Using Coloured Petri Nets", International Journal of Information Technology and Computer Science, 2014 Publication	1%
9	cas.et.tudelft.nl Internet Source	1%
10	convecs.inria.fr Internet Source	1%
11	medium.com Internet Source	<1%
12	"Martin Davis on Computability, Computational Logic, and Mathematical Foundations", Springer Science and Business Media LLC, 2016 Publication	<1%

Exclude quotes On
Exclude bibliography Off

Exclude matches Off