Taten H. Knight
L30052460
knight.taten@gmail.com

Homework Assignment 1
General Knowledge on Computer Science

2022-07-18

1 Inquiry 1

Week 1 of Research in Computer Science covered three core subjects: algorithm analysis, algorithm correctness, and software verification. These subjects are both fundamental to the study of computer science and remain relevant in the present and future progress of the field. I will briefly review the related literature for each topic and finish with a short summary and their relationship to each other and to the broader field of computer science.

1.1 Algorithm Analysis

Algorithm analysis is the process of examining an algorithm and determining its overall efficiency in memory and time, most commonly described in terms of *complexity*. For some function $f(x)$ what is the complexity C w.r.t. x. The most commonly used descriptor of complexity is big O notation, which describes the known worst possible behavior of the algorithm with respect to the input. θ notation describes the asymptotic behavior of the algorithm. For example, for the following 'algorithm' has a theta time complexity of $\theta(A)$ because the time it takes to run is proportional to the value of A:

```
A = 10
for a in [1:A]:
  print(a)
```

Space (memory) describes the memory taken up during the execution of the algorithm. The previous algorithm has space complexity of $O(N)$ because each element of the array must be stored and printed. There are others ways to describe the complexity of algorithms that have more strict criteria or describe other conditions, but the main point of algorithm analysis is to determine if use of the algorithm is viable in its environment, or if we need to use a more efficient solution to meet our constraints.

1.2 Algorithm Correctness

Determining algorithm correctness is determining if a given algorithm yields the correct results across all possible inputs, including handling invalid inputs appropriately. Correctness is paramount in safety-critical systems and important in all others to guarantee functionality. Absolute correctness is determined via the use of formal mathematical proofs. At the low level algorithm level this can be simple, but correctness is also needed at a high level for large software systems. To do this we can apply these proofs at a high level using software verification techniques.

1.3 Software Verification

Software verification is the process of determining whether software functions as intended. There are different standards for correctness and different methods of verifying the software. Verification can fall into three categories: peer review, testing, and formal methods. Peer review requires individuals manually reviewing code that has not been run, and only catches approximately 60% (median) of errors. There is testing which involves running human written tests for correct outcomes, and requires a large chunk of development time being devoted to writing the tests. Lastly there is formal verification, which uses formal mathematical methods to verify that the software only has outputs with the specified criteria. Although it is very intensive, the verification method is generally reusable after it has been created once for a given software system and can save many man hours for repeated verification. It can also provide better coverage, though it suffers the same drawbacks as other forms of verification; it cannot confirm the lack of errors, only the presence of them.

1.4 Summary

When developing software, the goal is to create software that performs the intended function, is correct (in that it responds appropriately to inputs with correct outputs), and is efficient enough to be used in a production environment. We can accomplish these goals through algorithm analysis, determining correctness on the individual algorithm level, and verifying software systems via peer review, testing, and formal verification. Although formal verification has not been easily achievable until relatively recently, some major companies such as Dassault-Aviation are using formal verification, testing, and peer-review in verifying portions of safety-critical systems, which indicates a shift away from past paradigms. This shift can result in better coverage and more long-term savings over a software development cycle.

2 Inquiry 2

Software Verification: Testing vs. Model Checking by D. Beyer and T. Lemberger, compares the current viability of automatic software testing against that of automatic software model checking. They concluded that software model checking is now mature enough to surpass automatic software testing in finding bugs in a program by comparing the results of both methods (six testing tools and four model checkers) on 5693 C programs. They also stress that testing should not be removed completely, but that model checking can outperform testing in some circumstances, and that a combination of methods can yield better overall results.

2.1 Definitions

Software testing is when a verification program is given a set of specifications and a wide set of input values, and we verify that the specifications hold across the test inputs. *Model Checking* explores the entire state space of the software and looks for violations of the given specification. Note that because the potential states of a program are essentially infinite we use an abstraction that simplifies or narrows the state space while still providing good coverage. ”A *violation witness* is an automaton that describes a set of paths through the program that contain a specification violation.”

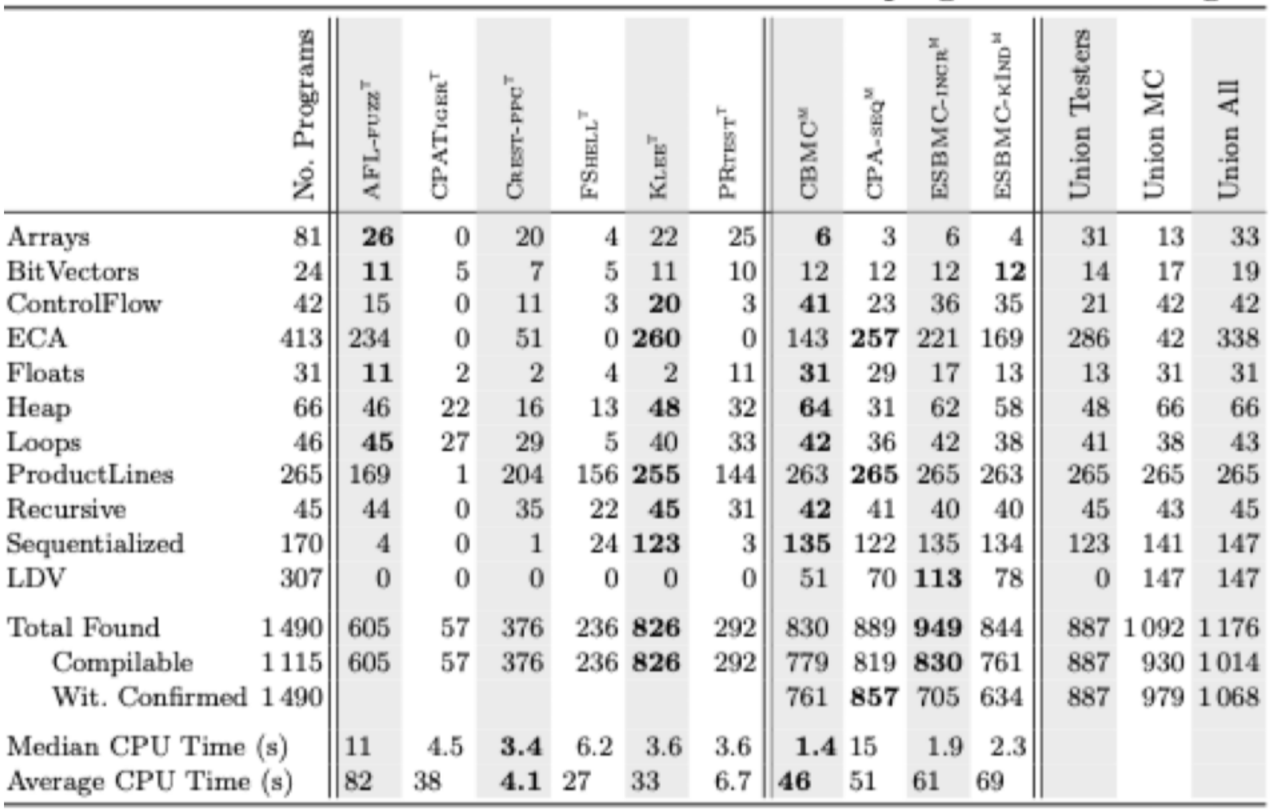
2.2 Methods

The study took four separate model checkers and 6 automatic software testers that each operated off of different principles and ran them against the 5693 separate C programs. They then took the results of these runs (the found bugs) and verified that they were correct using violation witnesses. They accomplished this by building a unifying framework for test best falsification that formats the source code of the C program, runs the test generator, extracts tests vectors, and checks if the tests cases exposed a bug.

2.3 Results

The results can be summarized by looking at the table from the study shown below:

Table 2: Results for testers and model checkers on programs with a bug



	No. Programs	AFL-fuzz^T	CPATiger^T	Crest-ppc^T	FShell^T	Klee^T	PRtest^T	CBMC^M	CPA-seq^M	ESBMC-incr^M	ESBMC-kInd^M	Union Testers	Union MC	Union All
Arrays	81	**26**	0	20	4	22	25	**6**	3	6	4	31	13	33
BitVectors	24	**11**	5	7	5	11	10	12	12	12	**12**	14	17	19
ControlFlow	42	15	0	11	3	**20**	3	**41**	23	36	35	21	42	42
ECA	413	234	0	51	0	**260**	0	143	**257**	221	169	286	42	338
Floats	31	**11**	2	2	4	2	11	**31**	29	17	13	13	31	31
Heap	66	46	22	16	13	**48**	32	**64**	31	62	58	48	66	66
Loops	46	**45**	27	29	5	40	33	**42**	36	42	38	41	38	43
ProductLines	265	169	1	204	156	**255**	144	263	**265**	265	263	265	265	265
Recursive	45	44	0	35	22	**45**	31	**42**	41	40	40	45	43	45
Sequentialized	170	4	0	1	24	**123**	3	**135**	122	135	134	123	141	147
LDV	307	0	0	0	0	0	0	51	70	**113**	78	0	147	147
Total Found	1 490	605	57	376	236	**826**	292	830	889	**949**	844	887	1 092	1 176
Compilable	1 115	605	57	376	236	**826**	292	779	819	**830**	761	887	930	1 014
Wit. Confirmed	1 490							761	**857**	705	634	887	979	1 068
Median CPU Time (s)		11	4.5	**3.4**	6.2	3.6	3.6	**1.4**	15	1.9	2.3			
Average CPU Time (s)		82	38	**4.1**	27	33	6.7	**46**	51	61	69			

We can see that all of the model checkers outperform the testers on total bugs found, and another result shows that the model checkers also have lower CPU times across the board. The only comparable tester is the KLEE in both CPU time and bugs found. Interestingly, each of the tests fail to catch different bugs, and the Union All column shows that the most bugs can be found by applying all model checkers and tests.

2.4 Conclusion

Based on the above results we see that if you are only able to choose a single verification method, a model checker is the best choice for finding the largest number of bugs as efficiently as possible. If, however, the goal is to find as many bugs as possible, combining multiple model checkers and testers will yield the highest number of bugs.