PROJECT REPORT FOR CPSC 513

FORMAL VERIFICATION USING THEOREM PROVING

December 29, 2018

Student ID: 19115724
University of British Columbia
Electrical and Computer Engineering

Contents

INTRODUCTION

0.0.1 What is formal verification?

High profile bugs continue to plague the software and the hardware community. This leads to major problems in the reliability, safety and security of systems. In my definition of software of verification, using the existing tools to write bug-free code, in order to prove the correctness of the program with respect to a mathematical specification is formal verification.

0.0.2 What was the motivation behind the project?

Using the tools and techniques of formal verification, I learnt how to specify the correct program behaviour, prove the correctness of my code, use decision procedures and model checkers to verify my code.
There are two types of correctness in the current scenario. There can be a situation where the algorithm is perfect. However, the fault can be in the implementation. Here is where proving correctness comes into picture. Firstly, the specification must be precise, meaning of code should be well-defined and finally the reasoning should be sound.

0.0.3 Approach

The bigger purpose about formal verification for systems scientists/engineers is to understand the system, and depending on the system modelling, the methodology can be selected. The bigger purpose is to systematically find bugs in order to ensure the safety, reliability and security of the systems.
The two main approaches are automated verifiers and using model checking.

1. In order to prove functional correctness of a system, we require specification of the system and the proof. In automated verifiers, for automated proofs, there are two methods

   - Automated theorem provers - Z3, CVC4 and Alt-Ergo
   - Interactive theorem provers - Coq, Isabelle etc

   Verifiers are very complex systems.

2. The second approach is model checking, which is a fully-automatic technique for finding bugs or proving their absence. The specifications are defined in propositional

temporal logic and the verification is followed by exhaustive state space search. This does not provide proofs instead provides diagnostic counterexamples. This does not provide proofs.
However, it leads to state-explosion. In order to deal with the state space explosion techniques we use techniques such as

- Bounded model checking
- Symbolic representations
- Abstraction and refinement

0.0.4 Problem statement

After having a couple of ways to approach software verification techniques, the software that I try to tackle as a part of the project is verifying equivalence between storage system driven applications. Suppose in an IoT based application where the sensor read and writes to MongoDB. However, the developer decides to change the database to MySQL due to performance issues. However, the user interface should not be changed. In order to verify such a situation, numerous test cases are required in order to make sure that the interface works the same way as before and the results do not change.

0.0.5 Glimpse

However, as I proceeded a number of lessons were waiting to be learned. One major lesson is: it needs deep understanding of maths and proofs in case of using theorem provers. Partial understanding will lead to partial verification which can be a waste.
In terms of theorem proving, the invariants should be defined carefully. When it comes to applying model checking, what becomes important are the abstractions and the correct mapping of the system.
I started by understanding the flow from propositional logic which moves to dynamic logic for theorem proving. Finally moving towards temporal logic for model checking. It's been an informative ride.

This can be useful for people entering into the world of formal verification and have no clue how to start digging in such a huge field. This will definitely help you get started.

LOGIC AND PROOF

In order to start using tools available such as Z3, CVC4 or any other standalone theorem provers, a basic understanding of how to use the tools is required. The basic structure of all the theorem provers is similar. No matter which automated verifier you decide to use, the ultimate goal remains the same. The interface varies however, you define your program with proper annotations and specifications.
The mathematicians had different ways of expressing information in the form of logic. The logic forms have different properties which become an important part of expressing the system and then proving it. It consists of syntax and semantics. The most crucial property to proving is soundness of the proof.

0.0.6 Propositional Logic (PL)

Propositional logic is the simplest form of logic. It enjoys a limited syntax which include conjunction, disjunction, negation, Exclusive or, Implication, Bi-implication. This syntax is supported semantically by an interpreted function I, which either tells us true or false. Since propositional logic is so simple, it includes completeness of PL and Decidability of PL.

0.0.7 First-order logic (FOL)

First order logic is an addition to propositional logic. It expresses behaviour that propositional logic is not capable of expressing. In addition to what propositional logic contains, it also contains, universal quantification and existential quantification a innumerable set of individual variable symbols. However, FOL is not decidable because it can have infinite number of interpretations of a formula. Hence an automated mechanism is required to verify inconsistent formula.

FOL provides more features to express, however, it is still a static logic. In order to verify a program, which changes states and consists of different states, and so its not necessary that at the end of every outcome the post and the preconditions are satisfied and in order to incorporate this, there is dynamic logic.

0.0.8 Dynamic logic

In order to provide more flexibility in representing programs. Dynamic logic provides modalities that talks about what is true after the program runs. DL grammer contains everything

propositional logic and FOL consists of. It also contains box modality and diamond modality in order to represent the program end.

0.0.9 Temporal Logic

Temporal logic is any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time. It's a logic for specifying properties over time. There are two main flavours of temporal logic mainly : Linear Temporal logic and Computation tree logic.

0.0.10 Conclusion from logic

So in general we have many different ways of expressing our system. A lot of programming languages are modelled based on the logics we defined above. However, when try to verify a system based on it's safety and liveness property. Or when we try to verify a system using theorem proving. It is important to understand the system first. Based on the functioning of the system, we can chose which logic will the most suitable to model our system. The paper [3] is a good paper to start for understanding the different logics out there for modelling the systems.

0.0.10.1 Tools available

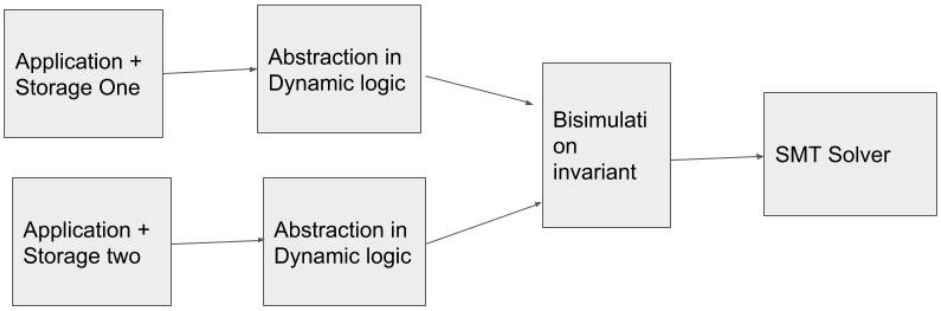
The different formal tools available to us have a properties language which is in one of the above mentioned forms and other forms not mentioned above like XTL, mu-calculus. Most of the tools are built in C, C++ and java.
For information on different sets of tools available [5] is a good place to start. This gives a brief overview of what sort of tools are available for what sort of systems.

MODELLING THE SYSTEM

In order to prove different systems using theorem proving, it is important to understand how to write programs and prove them mathematically. The Dijkstra paper [2] does a good job in getting started.
The first and the most basic program to try and prove is the GCD pogram. Considering the program, we can see that, dynamic logic would be the one which can easily help us represent and prove the program.
Based on this, I decided to use the tool called Why3 developed by inria which provides a strong platform to formally verify a system. It allows us to implement the system/algorithm in WhyML language which converts it to Ocaml code which can be run eventually.

0.0.11 Initial modelling

As the idea has been explained in the Introduction section of Problem statement. I considered two storage systems whose underlying data-structure is B+ trees and LSMs. The reason I chose these were because these two are the most commonly used Data structures in storage systems. There are others too but I decided to get started with Binary trees. In order to do so, Why3 initially seemed like a strong system that would help me achieve this.
The idea was represent the data structures for storage system A and storage system B. Using invariants, I decided to check if the read and the writes perform exactly the way they are supposed to perform. It was inspired by the work done by Microsoft research on checking equivalence of databases. [5]



0.0.12 Questions

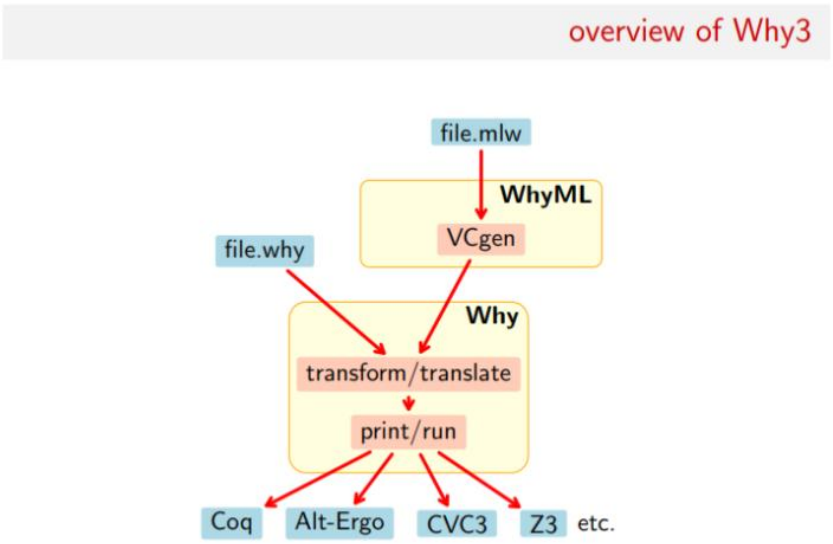
Based on my modelling and planning I wanted to understand how those things will work out for me. The main questions I was looking forward to answer were:

1. How can I model storage systems and compare their functionality based on the application?
2. Can I find a way to replace the fact that we have to run numerous benchmarks with the application when we change the storage system after N years by replacing it with this system where we don't have to run exhaustive tests
3. If we can come up with such a system, which tool will be the most useful in order to achieve it?
4. Is this a practical solution?

These were set of questions which I intended to answer by working on this project.
More that these questions, the project helped me get a better understanding of finding the right problems and understanding if they are really worth pursuing. Is it worth putting in so much effort for simple systems where regression testing can be an easier way to go.
So on the next page I will be talking about the awesome tool which I used and the experimentation which I conducted in order to understand if it is good enough for the system.

WHY3

Why3 provides an imperative programming language (with polymorphism, algebraic data types, pattern matching, exceptions, references, arrays, etc.), a specification language that is an extension of first-order logic, and a technology to verify programs using interactive and automated theorem provers (Coq, Alt-Ergo, Z3, CVC3, etc.) [4].



LEARNING WHYML

WhyML is a language similar to Ocaml. It is based out of Ocaml. In order to use theorem provers, learning a new programming language and the features it provides in order to prove or verify a system becomes necessary. In order to learn the language I tried different set of examples starting from basic absolute values.

0.0.13 Automated theorem provers

As I explained in the previous section Why3 provides support for using over 16 theorem provers. The default theorem prover is Alt-Ergo which is automatically installed when we install Why3. The list of theorem provers I used.

1. Alt-Ergo (default)
2. Z3
3. CVC4

Z3 and CVC4 provide support for equality, arithmetic, datatypes, bitvectors. Alt-Ergo does not provide support to all the theories that CVC4 and/or Z3 do.

0.0.14 Examples with WhyML

Important points which I learnt through this program:

1. How to import theories from Why3's standard library
2. How to annotate functions using the 'ensures' keyword
3. How to use the 'result' keyword when specifying a function
4. Using the ref keyword for for immutable values
5. Using ghost functions
6. Using variant in order to ensure that the loop ends.

The observations after learning to write the basic programs and running different solvers on them gave me different results.
When we run the theorem provers on our WhyML code, with annotations, the code is broken down into different sets of goals. These goals are proved step by step. Depending on the theories which the prover supports, it tried to prove the goals step by step.

0.0.14.1 Examples

1. Finding the absolute value of an integer, finding the sum of numbers in an array and then finding the maximum. Running this in the Why3 IDE, broke down the program into several number of goals and depending on the goals and running provers, it solved those goals.

2. Pattern matching is an important part of Ocaml and WhyML, which leads to building lists and then running pattern matching on them.

3. Apart from using whyML language it also allows us to use the logic language which allows us to solve problems such as Einstein problem. Where we define the predicates and the functions. It is not necessary for us to always use .mlw format in order to prove something.

4. After working with basic programs such as the ones mentioned above, I tried to implement data structures using whyML. I started with dynamic arrays and using the language skills I learnt above, I implemented dynamic arrays.

CONCLUSIONS

0.0.15 Answers to the Questions during modelling the system

1. How can I model storage systems and compare their functionality based on the application?
   Ans: In order to model the system, data structures being used to represent the system is a good approach. However, the problem here is, after modelling what functionality am I trying to prove by doing this. If I want to prove application based reads for two different storage systems, and writes. Just checking for equivalence is not enough.

2. Can I find a way to replace the fact that we have to run numerous benchmarks with the application when we change the storage system after N years by replacing it with this system where we do not have to run exhaustive tests?
   Ans: Yes I can but the question again comes down to is it worth spending effort into this? The benchmarks developed for each storage system work well enough to solve this problem.

3. If we can come up with such a system, which tool will be the most useful in order to achieve it?
   Ans: Why3 can be used as one of the systems, however, to build a full-fledged system, a language has to be developed which parses the application and finds the functionality of the storage system which is being used. Based on that it can check completely for those usage factors and try to prove it completely.

4. Is this a practical solution?
   Ans: Currently, the way my modelling is designed, it is not, but if someone really is able to do this, it can be very useful.

0.0.16 Advantages of using Why3

1. When the program is broken down into goals, each goal is proven correct one by one by the solver we use. However, when we are using one particular type of theorem prover, it necessarily does not have all theories. The other issue is that not SMT solvers have all theories strong. The non-linear arithmetic theory is comparitively stronger in Z3 than in CVC4.
   Depending on the strength of the theory we can use all the theorem provers in unison to prove our code. [6]

0.0.17 Lessons learnt

1. **Never follow the tool first system later approach.**

   To many this might seem obvious, however, I followed this approach and it hit me right back with bad results. The reason is, I first picked a tool and familiarizes myself with the tool Why3. However, the big problem was I tried to find a topic which will fit in well with this specific tool and towards the end I realized that it will not really work.

2. **Defending why using the formal tools gives you better results than using the normal testing tools or other approaches**

   One question that keeps coming back is when we find attacks, on a system, there can be multiple ways to automate it, one of the ways is using the formal tools. The question is how are they better than other available tools or methods.
   For example for modelling two databases based on application, why is theorem proving a better approach than the approaches available such as regression testing. These things should be very clearly defended.
   It is not necessary to use the formal tool. It should depend on what the application is asking for and what would be the easiest way to proceed with it.

REFERENCES

1) C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM, October 1969, pp. 576-583*

2) CEdsger W. Dijkstra, "Guarded Commands, Nondeterminacy, and the Formal Derivation of Programs," *Communications of the ACM, 18(8), August 1975, pp. 453-457.*

3) Frank Wolter and Michael Wooldridge, "Temporal and Dynamic Logic", *Department of Computer Science, University of Liverpool Liverpool L69 7ZF, UK*

4) Jean-Christophe FilliÃćtre 1, 2 Andrei Paskevich, " Why3 – Where Programs Meet Provers", ESOP'13 22nd European Symposium on Programming, Mar 2013, Rome, Italy. Springer, 7792, 2013, LNCS

5) YUEPENG WANG, ISIL DILLIG, SHUVENDU K. LAHIRI, WILLIAM R. COOK , " Verifying Equivalence of Database-Driven Applications", PLDI 2018

6) SMT Library http://smtlib.cs.uiowa.edu/

7) Model Checking tools https://en.wikipedia.org/wiki/List of model checking$_{t}ools$