Sitnikovski, Boro; Goracinova-Ilieva, Lidija; Stojcevska, Biljana

Article

Models for software verification: Proving program correctness

UTMS Journal of Economics

Provided in Cooperation with:
University of Tourism and Management, Skopje

Suggested Citation: Sitnikovski, Boro; Goracinova-Ilieva, Lidija; Stojcevska, Biljana (2021) : Models for software verification: Proving program correctness, UTMS Journal of Economics, ISSN 1857-6982, University of Tourism and Management, Skopje, Vol. 12, Iss. 1, pp. 32-39

This Version is available at:
https://hdl.handle.net/10419/281889

Boro Sitnikovski, Lidija Goracinova-Ilieva, and Biljana Stojcevska. 2021. Models for Software Verification: Proving program correctness. *UTMS Journal of Economics* 12(1): 32–39.

Original scientific paper
(accepted March 05, 2021)

MODELS FOR SOFTWARE VERIFICATION: PROVING PROGRAM CORRECTNESS

Boro Sitnikovski[1]
Lidija Goracinova-Ilieva
Biljana Stojcevska

Abstract:
The accuracy of computer systems represents the property that they are working as the users expect. Very often, these computer systems give inaccurate or wrong results. However, designing correct computer systems is a complex and expensive task. There are several ways to deal with this problem. In practice, the most common approach is to design and perform tests. However, these tests can only detect a specific set of problems. Another (more expensive) approach is to do a formal proof of correctness for a given code. This proof of correctness is, in fact, mathematical proof that the software works according to given specifications. Mathematical evidence covers all possible cases, and it is this evidence that confirms that code does exactly what it is intended to do. There are several platforms and mathematical models for software verification. Formal verification is based on mathematical proofs, and these platforms are divided into manual and automatic. Among the manual proof verification software, some of the most known ones are the programming languages Coq (based on type theory), Idris, etc. These are manual theorem provers, as the proof must be handwritten. Another family of theorem provers is the so-called automatic provers, which use algorithms to automatically deduce a given theorem. The programming language Dafny is one of their best representatives. This paper aims to show the state-of-the-art tools used today.

Keywords: software verification, software verification models, software verification platforms.

JEL classification: M42, M49

INTRODUCTION

Software is getting more complex and it's become a crucial part of everyday life. Users expect the systems to function at all times, according to their needs. However, in some cases the software may not function properly - this is called a software bug. Therefore, software bugs represent a fault in a computer system. As software grows every day in complexity, the probability that it will have bugs increases. Software bugs, depending on the problem that the software addresses, can cost millions. Therefore, it remains a crucial part to find and address most bugs before the software is pushed into production.

[1]**Boro Sitnikovski,** MSc.; **Lidija Goracinova-Ilieva,** Ph.D., Full Professor; **Biljana Stojcevska,** Ph.D., Associate Professor, Faculty of Informatics, University of Tourism and Management in Skopje, North Macedonia.

Manual testing of software to find bugs is one major area for testing. Another major area of software testing is formal verification – testing for correctness. Formal verification itself can be partially automated, but the bulk of the work remains manual.

In terms of formal verification, there is already a lot of research done by various institutions such as Microsoft Research and the French Institute for Research in Computer Science and Automation (INRIA). More specifically, the programming language Dafny is developed by Microsoft Research, and the programming language Coq is developed by INRIA. These two programming languages have one goal – to make it easier to detect bugs in software by using formal verification methods.

Several formal systems are usable for formal verification. One of them is known as Hoare logic, which the programming language is Dafny is based upon. Another commonly used formal system is type theory, which the programming language Coq is based upon.

1. THEORETICAL BACKGROUND

Formal verification's roots come from mathematics, and they date back to the beginning of the 20th century.

Russell's paradox represents the contradiction of the hypothesis $R \in R$ such that $R \in \{S: S \notin S\}$. In both cases $R \in R$ and $R \notin R$ – thus there is a contradiction.

As a response to Russell's paradox, Bertrand Russell himself proposed type theory as a solution (Russell, B., 1903). Type theory assigns a type to every term, ensuring that the problem of self-referencing sets is resolved with the usage of type hierarchies. This can be seen as the birth of type theory.

While working on discovering new foundations for mathematics, Alonzo Church introduced the lambda calculus in the 1930s (Church, A., 1936). Independently, a system with a similar power of expressiveness, the Turing machine, was introduced by Alan Turing in 1936 (Turing, A.M., 1936).

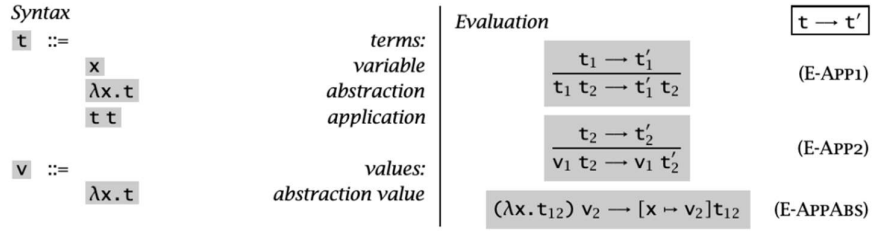

Figure 1: Syntax and evaluation rules of the lambda calculus (Pierce, B.C., 2002)

It was later shown that both lambda calculus and the Turing machine are Turing complete – they have the same computational power. Around the same period, it was also shown by Stephen Kleene and Kurt Gödel that general recursive functions have the same power of expressiveness (Kleene, S.C., 1936)

In later years, more advanced type theories were developed. More specifically, in 1940, Alonzo Church formulated the simply-typed lambda calculus (Church, A., 1940). This is a more restricted version of the original lambda calculus, and as such is not Turing complete.

Table 1: Type rules for the simply-typed lambda calculus

Rule Name	Formula
Var	$\frac{}{\Gamma, x{:}\,\tau \vdash x{:}\,\tau}$
Lam	$\frac{\Gamma, x{:}\,\tau_p \vdash t{:}\,\tau_r}{\Gamma \vdash (\lambda x : \tau_p . t){:}\,\tau_p \to \tau_r}$
App	$\frac{\Gamma \vdash t_f{:}\,\tau_p \to \tau_r \quad \Gamma \vdash t_p{:}\,\tau_p}{\Gamma \vdash t_f t_p{:}\,\tau_r}$

Between 1930 and 1970, one of the main developments concerning formal verification was the Curry-Howard correspondence. This correspondence represents the interpretation of proofs-as-programs and formulae-as-types (Wadler, P., 2015). That is, proofs are represented by computer programs, and the formula they prove is represented by their corresponding type. This correspondence shows the connection between computer programs and formal proofs, and as such it represents the building blocks of programming languages such as Coq.

Table 2: The Curry-Howard correspondence

Mathematics	Programming
Theorem	Type
Proof	Program
Correctness verification	Type checking
Cut elimination	Computation

In the meanwhile, in the late 1960s, Hoare logic was developed as a means to reason rigorously about the correctness of computer programs (Hoare, C.A.R. 1969). Hoare logic is independent of type theory and relies purely on mathematical representation – the Hoare triple – which is a triple that describes how a piece of code alters a state before and after its execution.

2. DAFNY/HOARE LOGIC

Dafny is a programming language that allows the expression of formal proofs with the usage of preconditions, postconditions, and invariants (Leino, K.R.M., 2010). As such, it is based on Hoare logic. It was developed by Microsoft in 2009 and is used in the academy and the industry.

As mentioned, the main component of Hoare logic is the Hoare triple $\{A\}\ B\ \{C\}$, in which it describes that before a command B executes, it has the state A – the precondition and that before a command B executes, it has the state C – the postcondition.

Table 3: Rules in the Hoare logic

Rule	Formula
Empty statement	$\frac{}{\{P\}\textbf{skip}\{P\}}$
Assignment	$\frac{}{\{P[E/x]\}x := E\{P\}}$
Composition	$\frac{\{P\}S\{Q\} \quad \{Q\}T\{R\}}{\{P\}\ S; T\ \{R\}}$
Conditional	$\frac{\{B \wedge P\}S\{Q\} \quad \{\neg B \wedge P\}T\{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$
Consequence	$\frac{P_1 \rightarrow P_2 \quad \{P_2\}S\{Q_2\} \quad Q_2 \rightarrow Q_1}{\{P_1\}S\{Q_1\}}$
While	$\frac{\{P \wedge B\}S\{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$

The Hoare rules are based on the Hoare triple. For example, the consequence rule specifies that given two programs, $\{p\}\ S_1\ \{r\}$ and $\{r\}\ S_2\ \{q\}$, we can compose them and conclude $\{p\}\ S_1;\ S_2\ \{q\}$. As an example, we can consider the following two programs:

$$\{x + 1 = 43\}\ y := x + 1\ \{y = 43\} \quad \text{Eq. 1}$$

$$\{y = 43\}\ z := y\ \{z = 43\} \quad \text{Eq. 2}$$

Using the Composition rule on Eq. 1 and Eq. 2, we can conclude the program in Eq. 3:

$$\{x + 1 = 43\}\ y := x + 1;\ z := y\ \{z = 43\} \quad \text{Eq. 3}$$

As we have shown, this pattern of reasoning is crucial to building correct programs. Given that Dafny is based on the same theory, we can construct the example program in Dafny.

Dafny will automatically verify the proof (See Figure 2: Composition in Dafny). It is sufficient that we only present the preconditions and the postconditions.

```
0 references
method first (x : nat) returns (y : nat)
  requires x + 1 == 43
  ensures y == 43
{
    y := x + 1;
}

0 references
method second (y : nat) returns (z : nat)
  requires y == 43
  ensures z == 43
{
    z := y;
}

0 references
method third (x : nat) returns (y : nat, z : nat)
  requires x + 1 == 43
  ensures z == 43
{
    y := x + 1;
    z := y;
}
```

Figure 2: Composition in Dafny

For automatic verification of proofs, Dafny relies on the Z3 theorem prover (De Moura, L., and Bjørner, N., 2008). Z3 is an efficient theorem prover – an SMT solver, developed by Microsoft in 2012. It has a wide range of applications both in the academy and in the industry.

$$a \wedge b = \neg(\neg a \vee \neg b) \qquad \text{Eq. 4}$$

Z3 uses a similar syntax to Lisp, and we will show a brief example of how Z3 can automatically prove the DeMorgan rule from Eq. 4:

```
(declare-const a Bool)
(declare-const b Bool)
(define-fun demorgan () Bool
    (= (and a b) (not (or (not a) (not b)))))
(assert demorgan)
(check-sat)
```

3. COQ/TYPE THEORY

Coq is a programming language that allows the expression of formal proofs with the usage of types (Barras, B. et al., 1997). As such, it is based on type theory. It was developed by INRIA in 1989 and it is heavily used both in the academy and in the industry.

As mentioned, the main component of type theory is that to each term, a type is assigned. For example, 3 belongs to the type of natural numbers Nat, and is represented as $3 : Nat$.

Table 4: The Brouwer–Heyting–Kolmogorov interpretation of mapping proofs to programs

Logic	Programming
Conjunction ∧	Product type
Disjunction ∨	Sum type
Implication →	Function type
Universal quantification ∀	Generalized product type Π
Existential quantification ∃	Generalized sum type Σ
True formula	Unit type
False formula	Bottom type

To understand how type theory may help with the automation of mathematical proofs, we will show an example.

Consider the product type $A \times B$ (the notation × represents the product type). Next, we will look at what kind of functions we can extract from this type.

One sensible function would be fst (short for "first"), with a corresponding type $A \times B \to A$.

$$fst : A \times B \to A, fst(a, b) = a \quad \text{Eq. 5}$$

Similarly, we can construct a function called snd (short for "second"), with a corresponding type $A \times B \to B$.

$$snd : A \times B \to B, snd(a, b) = b \quad \text{Eq. 6}$$

Using (composing) Eq. 5 and Eq. 6, we can construct a third function called $swap$ which is of type $A \times B \to B \times A$. In other words, the function accepts a pair of the form (a, b) where $a : A$ and $b : B$ and returns a pair of the form (b, a) where $a : A$ and $b : B$.

$$swap : A \times B \to B \times A, swap(e) = (snd(e), fst(e)) \quad \text{Eq. 7}$$

Thus, according to the Curry-Howard isomorphism, the function in Eq. 7 has a corresponding mathematical proof. Indeed, the type of $swap$ corresponds to the proof of the commutativity of ∧ (Eq. 8).

$$a \wedge b = b \wedge a \quad \text{Eq. 8}$$

The implementation of this proof in Coq is shown in Figure 3: Proof of the commutativity of logical "and".

```
Inductive natprod : Type :=
| pair : nat -> nat -> natprod.

Definition fst ( p : natprod ) :=
  match p with
  | pair x y => x
  end.

Definition snd ( p : natprod ) :=
  match p with
  | pair x y => y
  end.

Definition swap' ( p : natprod ) :=
  match p with
  | pair x y => pair y x
  end.

Notation "( x , y )" := (pair x y).

Theorem snd_fst_is_swap : forall (p : natprod),
  (snd p, fst p) = swap' p.
Proof.
  intros p.
  destruct p.
  simpl.
  reflexivity.
Qed.
```

Figure 3: Proof of the commutativity of logical "and"

The commands such as intros, destruct, simpl, reflexivity belong to the tactical language of Coq – this is a high-level language that gets translated to lambda calculus terms when the program is executed. The reason for the existence of the tactics language is that lambda terms can sometimes be unreadable. The tactics language can also provide proof automation.

CONCLUSION

This study summarized the state-of-the-art tools for software verification that are used today. We have shown that there are several theories for formal verification and that they differ in implementation and approach.

In this paper, presented were the programming languages Dafny and Coq, and there are many more that focus on formal verification and program correctness, to name a few: Agda, Idris, Lean.

Current research focuses on discovering new theories, as well as automating the construction of formal proofs and verification.

REFERENCES

Russell, B., 1903. *The Principles of Mathematics: Vol. 1.* Cambridge at the University Press, Cambridge, UK.

Church, A., 1936. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2), pp.345-363.

Turing, A.M., 1936. On computable numbers, with an application to the Entscheidungsproblem. *J. of Math*, 58(345-363), p.5.

Pierce, B.C., 2002. *Types and programming languages*. MIT press.

Kleene, S.C., 1936. General recursive functions of natural numbers. *Mathematische annalen*, 112(1), pp.727-742.

Church, A., 1940. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2), pp.56-68.

Wadler, P., 2015. Propositions as types. Communications of the ACM, 58(12), pp.75-84.

Hoare, C.A.R. 1969. An axiomatic basis for computer programming. *Communications of the ACM.* 12 (10): 576–580.

Leino, K.R.M., 2010, April. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning* (pp. 348-370). Springer, Berlin, Heidelberg.

De Moura, L. and Bjørner, N., 2008, March. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337-340). Springer, Berlin, Heidelberg.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C. and Parent, C., 1997. The Coq proof assistant reference manual: Version 6.1.